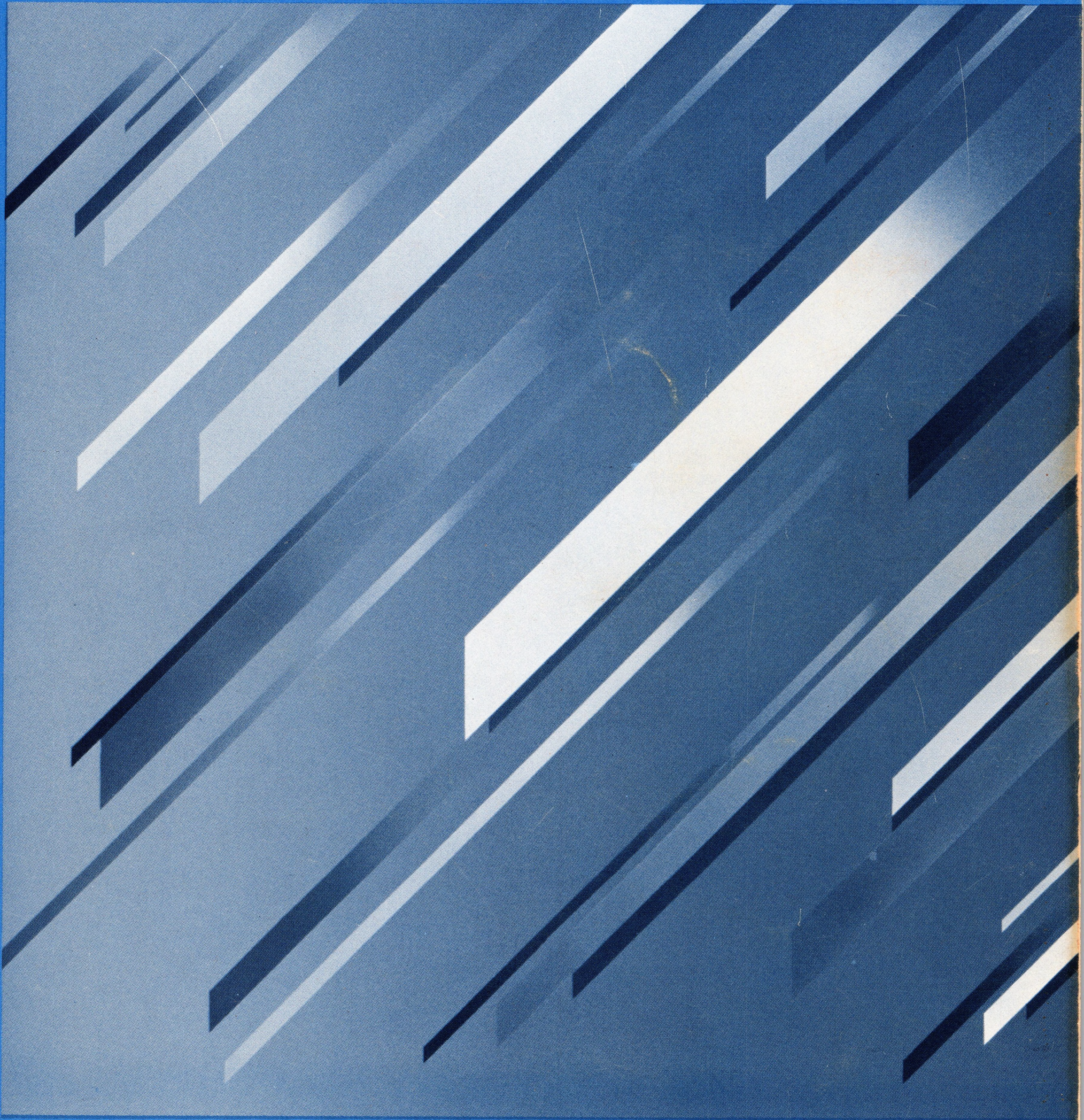


OS-9 Technical Information



TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK
COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A
RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL
STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS
AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

continued

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

OS-9 Technical Information

OS-9 Operating System: ©1983 Microware Systems
Corporation and Motorola Incorporated.
All Rights Reserved.
Licensed to Tandy Corporation.

OS-9 Technical Information:
©1983 Tandy Corporation
and Microware Systems Corporation.
All Rights Reserved.

UNIX is a trademark of Bell Laboratories.

TRS-80 is a registered trademark of Tandy Corporation.

Reproduction or use, without express written permission from
Tandy Corporation or Microware Systems Corporation of any
portion of this manual is prohibited. While reasonable efforts have
been taken in the preparation of this manual to assure its accuracy,
Tandy Corporation and Microware Systems Corporation assumes no
liability resulting from any errors or omissions in this manual, or
from the use of the information contained herein.

10 9 8 7 6 5 4 3 2 1

Introduction

OS-9 is a versatile multiprogramming/multitasking operating system for the TRS-80 Color Computer. It is well-suited for a wide range of applications. In addition to multiprogramming, its main features are:

- Comprehensive management of all system resources: memory, input/output, and CPU time
- Efficient operation in typical microcomputer configurations
- An expandable, device-independent unified I/O system

History and Design Philosophy

In the few years since its introduction, OS-9 has developed a worldwide user base of thousands. It is especially popular in the engineering, research, education, and scientific applications.

Microware and Motorola modeled OS-9 upon Bell Telephone Laboratories' UNIX operating system, which is becoming widely recognized as a standard for mini and micro multiprogramming operating systems. Although its implementation is different, OS-9 retains the overall concept and user interface of UNIX. If you are experienced with UNIX, you should feel quite comfortable with OS-9.

In addition, OS-9 introduces some important features to make the most of the 6809 microprocessor. Perhaps the most innovative of these is OS-9's "memory module" management system. This system provides extensive support for modular software techniques, which encourage simplified and more reliable programming.

The OS-9 modules are introduced in *Chapter 1* of this manual and are discussed in detail throughout.

About This Manual

This manual provides all the information necessary to install, maintain, expand, or write assembly-language programs for OS-9 systems. It assumes you are familiar with the 6809 architecture, instruction set, and assembly language.

Special Notations

For your convenience, the following special notations are used in this manual.

lower-case italics

represent words, letters, characters, or values that you supply.

\$nn

specifies that *nn* is a hexadecimal number. All other numbers in the text of this manual are in decimal (base 10) form, unless otherwise noted.

Contents

Chapter 1 / System Organization 1

Kernel, Clock Module, and INIT	2
Input/Output Modules	2
IOMAN	2
File Managers	2
Device Drivers	3
Device Descriptors	3
Shell	3
Boot	3

Chapter 2 / The Kernel 5

System Initialization	5
System Call Processing	6
OS9Defs and Symbolic Names	6
Types of System Calls	6
Memory Management	7
Memory Use	8
Multiprogramming	9
Process Creation	10
Process Termination	11
Process States	11
Execution Scheduling	12
Signals	12
Interrupt Processing	14
Physical Interrupt Processing	14
Logical Interrupt Polling System	15

Chapter 3 / Memory Modules 17

Module Types	17
Module Format	18
Module Header	18
Module Body	19
CRC Value	19

Module Headers: Standard Information	19
Module Headers: Type-Dependent Information.....	21
ROM Modules	23

Chapter 4 / OS-9's Unified Input/Output System 25

IOMAN	26
File Managers	26
Device Driver Modules.....	27
Device Descriptor Modules	28
Path Descriptors	31

Chapter 5 / Random Block File Manager ... 33

Logical and Physical Disk Organization	34
Identification Sector (LSN 0)	34
Disk Allocation Map Sector (LSN 1).....	35
File Sectors	35
File Descriptor Sector	35
Directories	37
RBFMAN Definitions of the Path Descriptor	37
RBF-Type Device Descriptor Modules	39
RBF-Type Device Driver Modules	40
RBFMAN Device Driver Subroutines	43

Chapter 6 / Sequential Character File Manager 53

SCFMAN Line Editing Functions	53
Read and Write	53
Read Line and Write Line	53
SCFMAN Definitions of the Path Descriptor	54

SCF-Type Device Descriptor Modules	57
SCF-Type Device Driver Modules	58
SCFMAN Device Driver Subroutines	60

Chapter 7 / Assembly Language Programming Techniques..... 67

How to Write Position-Independent Code	67
Addressing Variables and Data Structures	68
Stack Requirements	69
Interrupt Masks	69
Using Standard I/O Paths	69
Writing Interrupt-Driven Device Drivers	70
A Sample Program	72

Chapter 8 / System Calls 75

Calling Procedure	76
I/O System Calls	77
System Call Descriptions	77

Appendices 149

A / Alphabetical System Call Lists	149
B / Numerical System Call Lists	153
C / Memory Module Diagrams	157
D / Standard Floppy Disk Format	161
E / System Call Error Codes	163
F / Module and I/O Attributes	167

Table of Diagrams and Charts

Chapter 1

Color Computer OS-9 Modules	1
-----------------------------------	---

Chapter 2

Color Computer OS-9 Typical Memory Map	9
--	---

Chapter 3

Module Format	18
Module Headers: Standard Information	18
Type Codes	20
Language Codes	20
Executable Memory Module Format	22

Chapter 4

I/O System Modules	25
Branch Table Format	28
Device Descriptor Format	30
Path Descriptor: Standard Information	31

Chapter 5

Identification Sector	34
File Descriptor Sector	35
RBFMAN Definitions of the Path Descriptor	37
RBF-Type Device Descriptor Modules	39
RBF Device Memory Area Definitions	41
Drive Tables	42
Branch Table Format	43

Chapter 6

SCFMAN Definitions of the Path Descriptor 54

SCF-Type Device Descriptor Modules 57

SCF Device Memory Area Definitions 59

Branch Table Format 60

Chapter 7

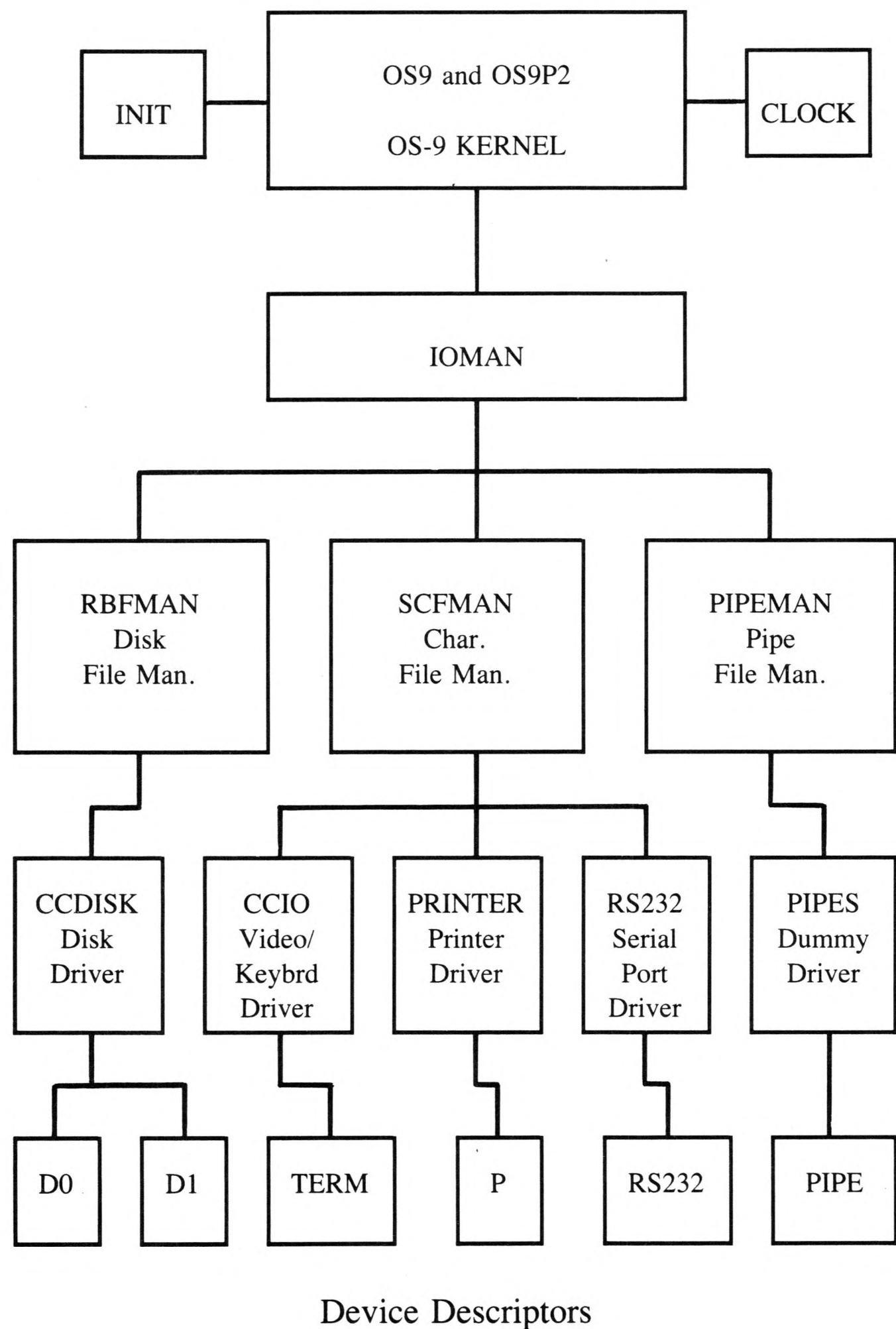
Use of Relative Addresses 67

Sample Program 72

1 / System Organization

OS-9 is composed of a group of modules, each of which has specific functions. The illustration below shows the major modules. Actual module names are capitalized.

Color Computer OS-9 Modules



Kernel, Clock Module, and INIT

The first level contains the “kernel,” “clock module,” and “INIT.”

The kernel provides basic system services, such as multitasking and memory management. It links all other system modules.

The clock module is a software handler for the specific real-time-clock hardware.

INIT is an initialization table used by the kernel during system startup. It loads initial tasks and specifies initial table sizes and initial system device names.

Input/Output Modules

The remaining modules make up OS-9's unified I/O system. They are defined briefly here and are discussed in detail in *Chapter 4*.

IOMAN

The second level (the level below the kernel) contains the “input/output” manager (IOMAN). IOMAN provides common processing of all I/O operations. It is required if any OS-supported I/O is to be performed.

File Managers

The third level contains the “file managers.” File managers perform I/O request processing for similar classes of I/O devices.

The Random Block File Manager (RBFMAN) processes all disk I/O operations. The Sequential Character File Manager (SCFMAN) handles all non-disk I/O operations that basically operate a character at a time, such as terminal and printer operations. The Pipe File Manager (PIPEMAN) handles pipes,

which are memory buffers that act like files for inter-process data transfers.

Device Drivers

The fourth level contains the “device drivers.” Device drivers handle basic physical I/O functions for specific I/O controller hardware. You may add your own customized drivers, or you may receive new drivers with accessory hardware devices.

Device Descriptors

The fifth level contains the “device descriptors.” These modules are small tables that associate each I/O port with its logical name, device driver, and file manager. They also contain the port’s initialization data and physical address.

When the device descriptors are used, only one copy of each driver is required for each type of I/O controller, regardless of the number of controllers the system uses.

Shell

The “shell” (not shown) is the command interpreter. It is technically a program and not part of the operating system itself, and is described fully in the *OS-9 Commands* manual.

Boot

All modules are loaded into RAM during system startup by the disk bootstrap module, “Boot.” Boot (not shown) is initially loaded into memory by the Color BASIC DOS command.

2 / The Kernel

The “nucleus” of OS-9 is the kernel, which supervises the system and manages system resources. It is about 3K bytes long and is contained in the OS9 and OS9P2 modules, with Boot, at memory addresses \$F000 - \$FEFF.

The kernel's main functions are:

- System initialization after reset
- System call processing
- Memory management
- Multiprogramming (6809 management)
- Interrupt processing

Note: Input/output functions are not included in the list because the kernel does not directly process them. Instead, it passes I/O system calls to IOMAN for processing.

System Initialization

After a hardware reset, the kernel initializes the system. This involves the following: locating modules loaded in memory from the OS-9 Boot file, allocating memory for internal tables, and running the system startup task (SYSGO). The INIT module is used during startup to specify initial table sizes and system device names.

The SYSGO program does the following:

1. Calls the shell and initializes the high-level system.
2. Starts the first user process.
3. Remains in the Wait state to make sure that all uses do not terminate and thus halt the system. SYSGO can keep the system going by restarting the first user process.

System Call Processing

“System calls” are used to communicate between OS-9 and assembly-language programs for such functions as memory allocation and process creation. In addition to I/O and memory management functions, system calls have other functions. These include interprocess control and timekeeping.

System calls use the SWI2 instruction followed by a constant byte representing the code. Parameters for system calls are usually passed in the 6809 register.

OS9Defs and Symbolic Names

A system-wide assembly-language equate file called OS9Defs defines symbolic names for all system calls. This file is included when assembling hand-written or compiler-generated code. The OS-9 assembler has a built-in macro to generate system calls. For example:

OS9 I\$read

is recognized and assembled as the equivalent to:

SWI2
FCB I\$read

Types of System Calls

System calls are divided into two categories, “I/O” calls and “function” calls.

I/O calls perform various input/output functions. Calls of this type are passed by the kernel to IOMAN for processing. The symbolic names for I/O calls begin with I\$. For example, the Read system call is called I\$read.

Function calls perform memory management, multi-programming, and other functions. Most are processed by the kernel. The symbolic names for function calls begin with F\$. For example, the Link system call is called F\$link.

The function calls include “user” calls and privileged “system mode” calls. (See *Chapter 8* for more information.)

Memory Management

Memory management is an important operating system function. Using memory modules, OS-9 manages the logical contents of memory and the physical assignment of memory to programs.

All programs that are loaded must be in the memory module format. This format allows OS-9 to maintain a module directory. The directory contains information about the module, including its name and address and the number of processes using it.

When a module no longer is needed, OS-9 deallocates its part of memory and removes its name from the module directory (except ROM, which is discussed later).

The memory modules are the foundation of OS-9’s modular software environment. These are some advantages of memory management:

- Automatic run-time “linking” of programs to libraries of utility modules
- Automatic “sharing” of reentrant programs
- Replacement of small sections of large programs for update or correction (even when in ROM).

Memory Use

OS-9 reserves some space at the top and bottom of RAM for its own use. The amount depends on the sizes of system tables that are specified in the INIT module.

All other RAM is pooled into a “free memory” space. As memory is allocated or deallocated, it is dynamically taken from or returned to this pool.

The basic unit of memory allocation is the 256-byte “page.” OS-9 always allocates memory in whole numbers of pages.

The data structure used to keep track of memory allocation is a 32-byte “bit map” located at \$0200 - \$021F. Each bit in this table is associated with a specific page of memory.

Cleared bits indicate that the page is free and available for assignment. Set bits indicate that the page is in use or that no RAM memory is present at that address.

Memory is allocated automatically when any of the following occurs:

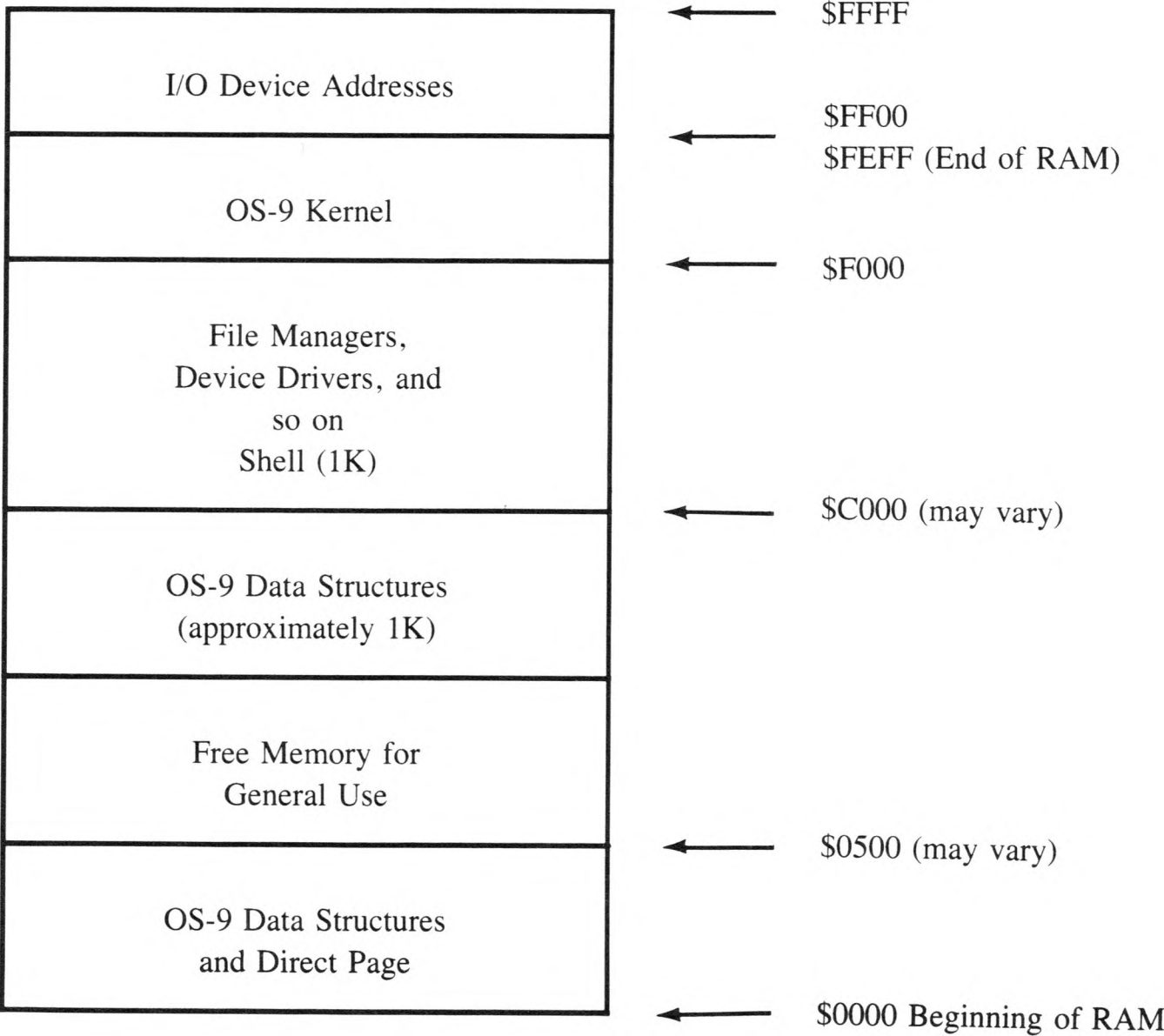
- Program modules are loaded into RAM
- Processes are created
- Process request additional RAM
- OS-9 needs I/O buffers or larger tables

Each of these functions usually has an inverse function that causes memory to be deallocated and returned to free memory.

In general, memory for program modules and buffers is allocated from high addresses downward. Memory for process data areas is allocated from lower addresses upward.

On the next page is a map of a “typical” system. Actual memory sizes and addresses may vary depending on the exact system configuration.

Color Computer OS-9 Typical Memory Map



Multiprogramming

OS-9 is a multiprogramming operating system. This means several independent programs called “processes” can be executed at the same time. By issuing the appropriate system call to OS-9, each process can have access to any system resource.

Multiprogramming functions use a hardware real-time clock that generates interrupts at a regular rate of 60 times per second. Therefore, 6809 time usually is divided into periods of 16.67 milliseconds. These periods are called “ticks.”

“Active” processes (those not waiting for some event) are run for a specific system-assigned period called a “time slice.” The length of the time slice depends on a process’ priority relative to all other active processes. Many OS-9 system calls are available to create, terminate, and control processes.

Process Creation

A process is created when an existing process executes a Fork system call (F\$fork). This call’s main argument is the name of the program module that the new process is to execute first (the “primary module”).

Find the Module. OS-9 first attempts to find the module in the module directory. If it does not find the module, OS-9 usually attempts to load into memory a mass-storage file, giving it the requested module name as a filename.

Assign a Process Descriptor. Once OS-9 finds the module, it assigns the process a data structure called a “process descriptor.” This is a 64-byte package that contains information about the process, its state, memory allocations, priority, queue pointers, and so on.

OS-9 automatically initializes and maintains the process descriptor. The process itself cannot access the descriptor; it has no need to do so.

Allocate RAM. The next step is allocation of RAM for the process. The primary module’s header contains a storage size. OS-9 uses this size unless the Fork system call asked for a larger area. OS-9 then attempts to allocate a contiguous memory area of the specified size from the free memory space.

Create or Abort. If OS-9 can perform all of the previous steps, it adds the new process to the active process queue for execution scheduling. If it cannot, it aborts the creation. The process that originated the Fork is informed of the error.

Assign Process ID and User ID. OS-9 assigns the new process a unique number called a “process ID.” Other processes can communicate with the process by referring to its ID in various system calls.

The process also has a “user ID,” which is used to identify all processes and files belonging to a particular user. The user ID is inherited from the parent process.

Process Termination

A process terminates when it executes an Exit system call (F\$exit) or when it receives a fatal signal. The termination closes any open paths, deallocates memory used by the process, and unlinks the primary module.

Process States

At any instant a process can be in one of three states:

- Active (The process is ready for execution.)
- Waiting (The process is suspended until a “child process” terminates or a signal is received.)
- Sleeping (The process is suspended for a specific period of time or until a signal is received.)

Each state has its own queue, a linked list of “descriptors” of processes in that state. State changes are performed by moving a process descriptor to another queue.

The Active State. Each active process is given a time slice for execution, according to its priority relative to all other active processes. The scheduler, which is in the kernel, makes sure that all active processes, even those of low priority, get some CPU time.

The Wait State. This state is entered when a process executes a Wait system call (F\$wait). The process remains suspended until one of its “child” processes dies or until it receives a “signal.” (See the “Signals” section below.)

The Sleeping State. This state is entered when a process executes a Sleep system call (F\$sleep), which specifies the number of ticks for which the process is to remain suspended. The process remains asleep until the specified time has elapsed or until it receives a signal.

Execution Scheduling

The OS-9 scheduler uses an algorithm that ensures that all active processes get some execution time.

All active processes are members of the “active process queue,” which is kept sorted by process “age.” Age is a count of how many process switches have occurred since the process’ last time slice. When a process is moved to the active process queue from another queue, its age is set according to its priority — the higher the priority, the higher the age.

Whenever a new process becomes active, the ages of all other active processes are incremented. When the executing process’ time slice has elapsed, the scheduler selects the next process to be executed (the one with the next highest age, the first one in the queue). At this time the ages of all other active processes are incremented. (Ages are never incremented beyond 255, however.)

An exception is a newly active process that was deactivated while in the system state. Such a process is given higher priority because it usually is executing critical routines that affect shared system resources. Therefore, it could be blocking other unrelated processes.

When there are no active processes, the kernel sets itself up to handle the next interrupt and then execute a Cwai instruction, which decreases interrupt latency time.

Signals

A signal is an asynchronous control mechanism used for inter-process communication and control. It behaves like a software interrupt (it can cause a process to suspend a program, execute a specific routine, and afterward return to the interrupted program).

Signals can be sent from one process to another process by the Send system call (F\$send). Or, they can be sent from OS-9 service routines to a process.

The signal can convey status information in the form of a 1-byte numeric value. Some of the signal “codes” (values) are predefined, but you may define most. The signal codes are:

- 0 = Kill (aborts the process; is non-interceptable)
- 1 = Wakeup (wakes up a sleeping process)
- 2 = Keyboard abort
- 3 = Keyboard interrupt
- 4 = User-defined (255)

When a signal is sent to a process, the signal is noted and saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. When it gets its next time slice, the signal is processed.

What happens next depends on whether or not the process has set up a “signal intercept trap” (signal service routine) by executing an Intercept system call (F\$icpt).

If it has set up a signal intercept trap, the process resumes execution at the address given in the Intercept call. The signal code is passed to this routine, which should terminate with an RTI instruction to resume normal execution of the process.

Note: A wakeup signal activates a sleeping process. It **does not** vector through the intercept routine.

If it has not set up a signal intercept trap, the process is aborted immediately. It is also aborted if the signal code is zero. If the process is in the system mode, the abort is deferred (it dies upon return to the user state).

A process may have a signal pending (usually because it has not been assigned a time slice since the signal was received). If it does, and another process tries to send it another signal, the new signal is aborted, and the Send system call returns an error. The sender should then execute a Sleep system call for a few ticks before trying to send the signal again. This gives the destination process time to process the pending signal.

Interrupt Processing

Interrupt processing is another important function of the kernel. OS-9 sends each hardware interrupt to a specific address. This address, in turn, specifies the address of the device service routine to be executed. This is called “vectoring” the interrupt. The address that points to the routine is called the “vector.” It has the same name as the interrupt.

Physical Interrupt Processing

Addresses \$FFF0 through \$FFFF (in the Color BASIC ROM) contain the hardware vectors required by 6809. Each address points to a RAM vector, as follows:

Interrupt	Vector
SWI3 (Software Interrupt 3)	\$0100
SWI2 (Software Interrupt 2)	\$0103
FIRQ (Fast Interrupt Request)	\$010F
IRQ (Interrupt Request)	\$010C
SWI (Software Interrupt)	\$0106
NMI (Non-Maskable Interrupt)	\$0109

OS-9 initializes each of these RAM vectors to point to a specific service routine in the kernel.

Software Interrupts (SWI, SWI2, and SWI3.) The software interrupts are vectored to user-definable addresses, which are local to each procedure. SWI2, however, usually is used for OS-9 system calls.

The SWI, SWI2, and SWI3 vectors point to routines that read the corresponding pseudo vector from the process’ descriptor and dispatch to it. This is why the Set SWI system call (F\$sswi) is local to a process; it only changes a pseudo vector in the process descriptor.

FIRQ Interrupt. The system uses the FIRQ interrupt. The FIRQ vector, which points to an RTI instruction, is not available to you.

IRQ Interrupt. Most OS-9 I/O devices generate IRQ interrupts. The IRQ vector points to the real-time clock and keyboard scanner routines. These routines, in turn, jump to a special IRQ polling system which determines the source of the interrupt. The polling system is discussed in the next section, “Logical Interrupt Polling System.”

NMI Interrupt. The system uses the NMI interrupt. The NMI vector, which points to the disk driver interrupt service routine, is not available to you.

Logical Interrupt Polling System

Because most OS-9 I/O devices use IRQ interrupts, OS-9 includes a sophisticated polling system. The IRQ polling system automatically identifies the source of the interrupt, and then executes its associated user- or system-defined service routine.

The Polling Table. The information required for IRQ polling is maintained in a data structure called the “IRQ polling table.” The table has a 9-byte entry for each device that might generate an IRQ interrupt. The table size is permanent and is defined by an initialization constant in the INIT module.

Each entry in the polling table is given a number from 0 (lowest priority) to 255 (highest priority). In this way, the more important devices (those that have a higher interrupt frequency) can be polled before the less important ones.

Each entry has 6 variables:

- **Polling Address** — points to the device’s status register. The register must have a bit or bits that indicate it is the source of an interrupt.
- **Flip Byte** — selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).
- **Mask Byte** — selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.

-
- **Service Routine Address** — points to the device's interrupt service routine. You supply this address.
 - **Static Storage Address** — points to the permanent storage area required by the device service routine. You supply this address.
 - **Priority** — sets the order in which the devices are to be polled (a number from 0 to 255).

Polling the Entries. When an IRQ interrupt occurs, the polling system is entered via the corresponding RAM interrupt vector. It starts polling the devices in order. Each entry's status register address is loaded into Accumulator A, using the device address from the table.

Then, an Exclusive-OR operation using the flip byte is executed, followed by a Logical-AND operation using the mask byte. If the result is non-zero, the device is assumed to be the cause of the interrupt.

The device's memory address and service routine address are read from the table and executed.

Note: The interrupt service routine should terminate with an RTS instruction, **not** an RTI instruction.

Adding Entries to the Table. Using the Set IRQ system call (F\$irq), you can make entries to the IRQ polling table. Set IRQ is a privileged system call, which can be executed only when OS-9 is in system mode (which is the case when device drivers are executed).

Note: The actual code for the interrupt polling system is located in the IOMAN module. The kernel OS9 and OS9P2 modules contain the physical interrupt processing routines.

3 / Memory Modules

In *Chapter 2*, you learned that OS-9 is based on the concept that memory is modular. This means that each program is considered to be an individual, named “object.”

You also learned that all objects that are loaded into memory must be in the module format. This format lets OS-9 manage the logical contents of memory, as well as the physical contents. Module types and formats are discussed in detail in this chapter.

Module Types

There are several types of modules, each of which has a different use and function. These are the main requirements of a module:

- It cannot modify itself.
- It must be position-independent so OS-9 can load or relocate it wherever space is available. In this respect, the module format is the OS-9 equivalent of “load records” used in older operating systems.

A module need not be a complete program or even 6809 machine language. It may contain BASIC09 “I-code,” constants, single subroutines, and subroutine packages.

Module Format

Each module has three parts; a “module header,” a “module body,” and a “cyclic-redundancy-check value” (CRC value).

Module Format

Module Header
Program or Constants
CRC Value

Module Header

At the beginning of the module (lowest address) is the module header. Its form depends upon the module’s use.

The header contains information about the module and its use. This includes the following:

- Size
- Type (machine language, BASIC09 compiled code, and so on)
- Attributes (executable, reentrant, and so on)
- Data storage memory requirements
- Execution starting address

Usually, you don’t need to write routines to generate the modules and headers. Most OS-9 programs — including BASIC09, Pascal, C, and the assembler — do this automatically.

Module Body

The module body contains the program or constants. It usually is pure code. The module name string is included somewhere in this area.

CRC Value

The last three bytes of the module are the Cyclic Redundancy Check (CRC) value. The CRC value is used to verify the integrity of a module.

The 24-bit CRC is performed over the entire module from the first byte of the module header to the byte just before the CRC itself. The CRC polynomial used is \$800FE3.

As with the header, you usually don't need to write routines to generate the CRC value. Most OS-9 programs do this automatically.

Module Headers: Standard Information

The first nine bytes of all module headers are defined as follows:

Relative Address	Use
\$00,\$01	Sync bytes (\$87,\$CD)
\$02,\$03	Module size
\$04,\$05	Offset to module name
\$06	Module type / Language type
\$07	Attributes / Revision level
\$08	Header check

Sync Bytes. Specify the location of the module. (The first sync byte is the start of the module.) These two bytes are constant.

Module Size. Specify the size of the module in bytes (includes CRC).

Offset to Module Name. Specifies the address of the module name string relative to the start of the module. The name string can be located anywhere in the module and consists of a string of ASCII characters having the most significant bit set on the last character.

Type/Language Byte. Specifies the type and language of the module.

The four most significant bits of this byte indicate the type. Eight types are pre-defined. Some of these are for OS-9's internal use only. The type codes are given here:

Code	Module Type
\$01	Program module
\$02	Subroutine module
\$03	Multi-module (for future use)
\$04	Data module
\$05-\$0B	User-definable
\$0C	OS-9 system module
\$0D	OS-9 file manager module
\$0E	OS-9 device driver module
\$0F	OS-9 device descriptor module

The four least significant bits of this byte indicate the language. The language codes are given here:

Code	Language
\$00	Data (non-executable)
\$01	6809 object code
\$02	BASIC09 I-code
\$03	PASCAL P-code
\$04	COBOL I-code
\$05-\$15	Reserved for future use

By checking the language type, high-level language run-time systems can verify that a module is the correct type before attempting execution. BASIC09, for example, may run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

Attributes / Revision Byte. Specifies the attributes and revision level of the module.

The four most significant bits of this byte are reserved for module attributes. Currently, only Bit 7 is defined. When set, it indicates the module is reentrant and therefore “sharable.”

The four least significant bits of this byte are a revision level from 0 to 15. If two or more modules have the same name, type, language, and so on, OS-9 keeps in the module directory only the module having the highest revision level. Therefore, you can replace or patch a ROM module, simply by loading a new, equivalent module that has a higher revision level.

Note: A previously linked module cannot be replaced until all processes linked to it have unlinked it (after its link count goes to zero).

Header Check. Specifies the one’s complement of the Exclusive-OR of the previous eight bytes.

Module Headers: Type-Dependent Information

More information usually follows the first nine bytes of the header. The layout and meaning vary, depending on the module type.

Module types \$0C-\$0F (system module, file manager module, device driver module, and device descriptor module) are used by OS-9 only. Their formats are given later in the manual.

Module types \$01 through \$0B have the general-purpose “user” format (executable format) shown on the next page. This format is used often for OS-9 programs that are called by Fork or Chain (F\$fork and F\$chain).

Executable Memory Module Format

Relative Address	Use	Check Range	
\$00	Sync Bytes (\$87CD)	header parity	
\$01			
\$02	Module Size (bytes)		
\$03			
\$04	Module Name Offset		
\$05			
\$06	Type	Language	module CRC
\$07	Attributes	Revision	
\$08	Header Parity Check		module CRC
\$09	Execution Offset		
\$0A	Permanent Storage Size		module CRC
\$0B			
\$0C	(Additional optional header extensions located here)		module CRC
\$0D			
			module CRC
	Module Body object code, constants, and so on		
	CRC Check Value		module CRC

As you can see from the chart on the previous page, this module format has two extra bytes in its header. They are:

\$09,\$0A	Execution offset
\$0B,\$0C	Permanent storage size

Execution Offset. The program or subroutine's offset starting address, relative to the first byte of the sync code. A module that has multiple entry points (such as cold start and warm start) may have a branch table starting at this address.

Permanent Storage Size. The minimum number of bytes of data storage required to run. Fork and Chain use this number to allocate a process' data area.

If the module will not be directly executed by a Fork or Chain system call (for instance a subroutine package), this entry is not used by OS-9. It is commonly used to specify the maximum stack size required by reentrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

ROM Modules

When OS-9 starts after a system reset, it searches the entire memory space for ROM modules. It finds them by looking for the module header sync code (\$87,\$CD).

When this byte pattern is detected, OS-9 checks the header to see if it is correct. If it is, the system obtains the module size from the header and performs a 24-bit CRC over the entire module. If the CRC matches, OS-9 considers the module to be valid and enters it into the module directory. All ROM modules that are present in the system at startup are automatically included in the system module directory.

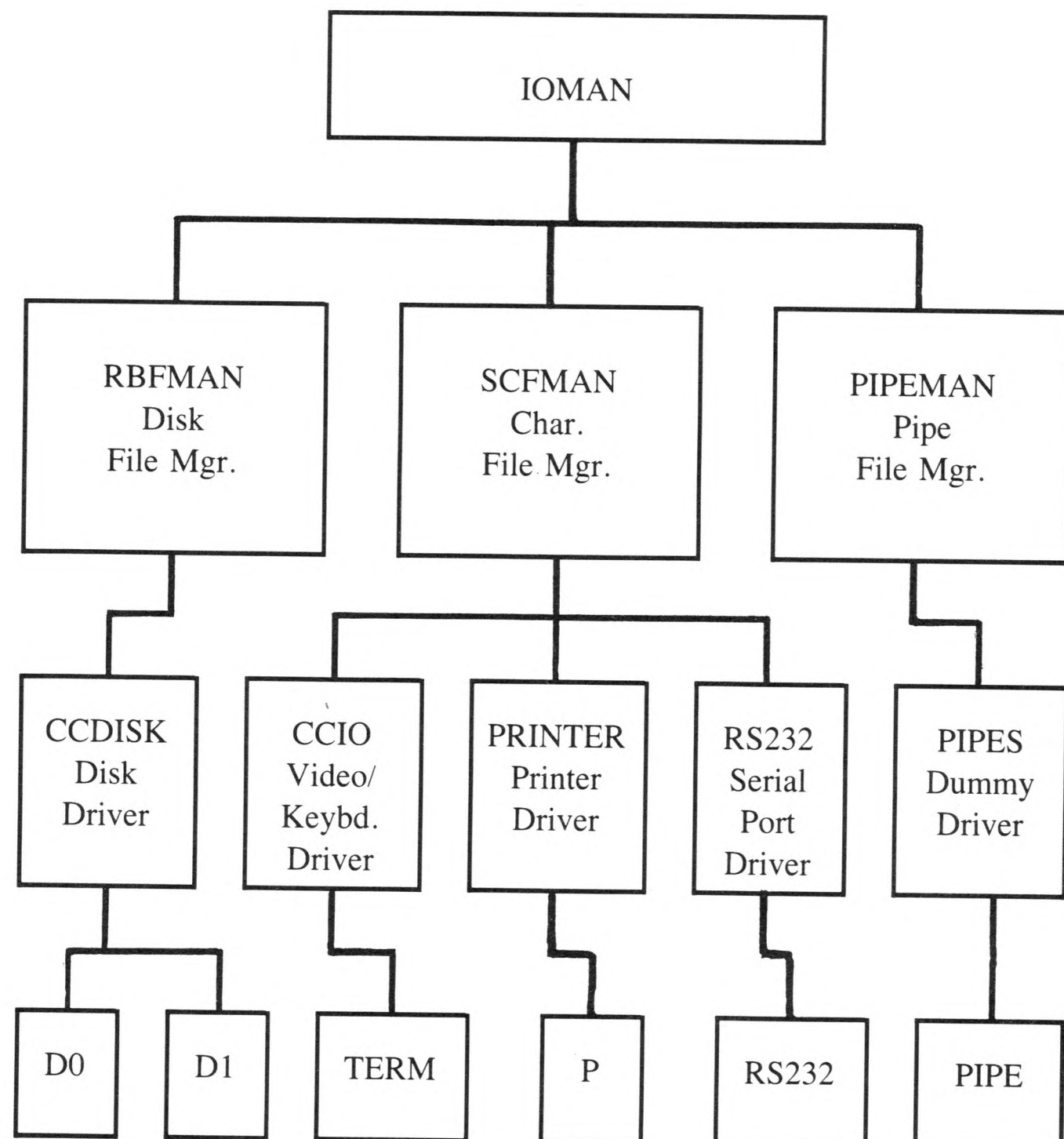
After the module search, OS-9 links to the component modules it found. This is the secret to OS-9's ability to adapt to almost any 6809 computer. It automatically locates its required and optional component modules and rebuilds the system each time it is started.

At startup, OS-9 also searches ROM for non-system modules. It locates any software you supply and, if the module's header and CRC information is correct, enters it into the module directory.

4 / OS-9's Unified Input/Output System

In Chapter 1, we mentioned that OS-9 has a unified I/O system, consisting of all modules except those on the kernel level. This chapter discusses the I/O modules in detail.

I/O System Modules



Device Descriptors

The I/O system provides system-wide, hardware-independent I/O services for user programs and OS-9 itself. All I/O system calls are received by the kernel and passed to IOMAN for processing.

IOMAN performs some processing, such as the allocation of data structures for the I/O path. Then, it calls the file managers and device drivers to do most of the work. Additional file manager, device driver, and device descriptor modules can be loaded into memory from files and used while the system is running.

IOMAN

IOMAN provides the first level of service of I/O system calls. It routes data on I/O process paths to or from the appropriate file managers and device drivers.

IOMAN also maintains two important internal OS-9 data structures — the “device table” and the “path table.” Never modify IOMAN.

When a path is opened, IOMAN tries to link to a memory module that has the device name given or implied in the pathlist. This module is the device descriptor. It contains the names of the device driver and file manager for the device. IOMAN saves the names so later system calls can be routed to these modules.

File Managers

OS-9 can have any number of file manager modules. Each of these modules processes the raw data stream to or from a class of device drivers that have similar operational characteristics. It removes as many unique characteristics as possible from I/O operations. Thus, it assures that similar devices conform to the OS-9 standard I/O and file structure.

The file manager also is responsible for mass storage allocation and directory processing, if these are applicable to the class of devices it served.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They may also monitor and process the data stream, for example, adding line-feed characters after carriage-return characters.

The file managers are reentrant. The two standard OS-9 file managers are:

- Random Block File Manager (RBFMAN) — supports random-access, block-structured devices such as disk systems and bubble memories. (*Chapter 5* discusses RBFMAN in detail.)
- Sequential Character File Manager (SCFMAN) — supports single-character-oriented devices, such as CRT or hardcopy terminals, printers, and modems. (*Chapter 6* discusses SCFMAN in detail.)

Device Driver Modules

The device driver modules are subroutine packages that perform basic, low-level I/O transfers to or from a specific type of I/O device hardware controller. These modules are reentrant so one copy of the module can run at the same time several devices that use identical I/O controllers.

Device driver modules use a standard module header, in which the module type is specified as code \$0E (device driver). The execution offset address in the module header points to a branch table that has a minimum of six 3-byte entries.

Each entry is typically a LBRA to the corresponding subroutine. The file managers call specific routines in the device driver through this table, passing a pointer to a path descriptor and the hardware control register address in the 6809 register. The branch table looks like this:

Code	Meaning
+ \$00	Device initialization routine
+ \$03	Read from device
+ \$06	Write to device
+ \$09	Get device status
+ \$0C	Set device status
+ \$0F	Device termination routine

(For a complete description of the parameters passed to these subroutines, see the “Device Driver Subroutines” sections in *Chapters 5 and 6*.)

Device Descriptor Modules

Device descriptor modules are small, non-executable modules. They provide information that associates a specific I/O device with its logical name, hardware controller address(es), device driver, file manager name, and initialization parameters.

Unlike the device drivers and file managers — which operate on classes of devices — each device descriptor tailors its functions to a specific device. Each device must have a device descriptor.

Device descriptor modules use a standard module header, in which the module type is specified as code \$0F (device descriptor). The name of the module is the name by which the system and user know the device (the device name given in pathlists).

The rest of the device descriptor header consists of the information in the following chart:

Relative Address(es)	Use
\$09,\$0A	File manager name string relative address
\$0B,\$0C	Device driver name string relative address
\$0D	Mode/Capabilities: D S PE PW PR E W R (directory, sharable, public execute, public write, public read, execute, write, read)
0E\$, \$0F,\$10	Device controller absolute physical (24-bit) address
\$11	Number of bytes (n bytes) in the initialization table
<u>\$12,\$12 + n</u>	<u>Initialization table</u>

When a path to the device is opened, the initialization table is copied into the “option section” (PD.OPT) of the path descriptor. (See “Path Descriptors” in this chapter.)

The values in this table may be used to define the operating parameters that are changeable by the Get Status and Set Status system calls (I\$getstt and I\$setstt). For example, a terminal’s initialization parameters define which control characters are used for such functions as backspace and delete.

The initialization table may be up to 32 bytes long. If the table is less than 32 bytes long, the remaining values in the path descriptor are set to 0.

You may wish to add devices to your system. If a similar device controller already exists, all you need to do is add the new hardware and load another device descriptor. Device descriptors can be in ROM or loaded into RAM from mass-storage files while the system is running.

The diagram on the next page illustrates the device descriptor format:

Device Descriptor Format

Relative Address	Use	Check Range
\$00	Sync Bytes (\$87CD)	header parity
\$01		
\$02	Module Size	
\$03		
\$04	Offset to Module Name	
\$05		
\$06	\$F (Type) \$1 (Lang)	
\$07	Attributes Revision	
\$08	Header Parity Check	module CRC
\$09	Offset to File Manager Name String	
\$0A		
\$0B	Offset to Device Driver Name String	
\$0D	Mode Byte	
\$0E	Device Controller Absolute Physical Addr. (24 bit)	
\$0F		
\$10		
\$11	Initialization Table Size	
\$12, 12 + n	(Initialization Table)	
	(Name Strings, and so on)	
	CRC Check Value	

Path Descriptors

Every open path is represented by a data structure called a path descriptor (PD). It contains the information the file managers and device drivers require to perform I/O functions.

PDs are 64 bytes long and are dynamically allocated and deallocated by IOMAN as paths are opened and closed.

They are internal data structures, which are not normally referenced from user or applications programs. The description of PDs is presented here mainly for those programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections. The first 10-byte section is the same for all file managers and device drivers, as shown in the chart on the next page:

Path Descriptor: Standard Information

Name	Relative Address	Size (Bytes)	Use
PD.PD	\$00	1	Path number
PD.MOD	\$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	\$02	1	Number of open images (paths using this PD)
PD.DEV	\$03	2	Address of the associated device table entry
PD.CPR	\$05	1	Current process ID
PD.RGS	\$06	2	Address of the caller's register stack
PD.BUF	\$08	2	Address of the 256-byte data buffer (if used)

PD.FST	\$0A	22	Defined by the file manager
PD.OPT	\$20	32	Reserved for the Getstat/Setstat options

PD.FST is reserved for and defined by each type of file manager for file pointers, permanent variables, and so on.

PD.OPT is used as an option area for file or device operating parameters that are dynamically alterable. When the path is opened, IOMAN initializes these variables by copying the initialization table that is in the device descriptor module. User programs can change them later, using the Get Status and Set Status system calls.

PD.FST and PD.OPT are defined for the file manager in the assembly-language equate file (SCFDefs for SCFMAN and RBFDefs for RBFMAN).

5/ Random Block File Manager

The Random Block File Manager (RBFMAN) supports disk storage. It is a reentrant subroutine package called by IOMAN for I/O system calls to random-access devices. It maintains the logical and physical file structures.

During normal operation, RBFMAN requests allocation and deallocation of 256-byte data buffers. Usually, one is required for each open file. When physical I/O functions are necessary, RBFMAN directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks (pages).

RBFMAN does not deal directly with physical addresses such as tracks and cylinders. Instead, it passes to the device drivers address parameters, using a standard address called a “logical sector number”, or “LSN.” LSNs are integers from 0 to $n-1$, where n is the maximum number of sectors on the media. The driver translates the logical sector number to actual cylinder/track/sector values.

Because RBFMAN supports many devices that have different performance and storage capacities, it is highly parameter-driven. The physical parameters it uses are stored on the media itself.

On disk systems, this information is written on the first few sectors of Track 0. The device drivers also use this information, particularly the physical parameters stored on Sector 0. These parameters are written by the Format program that initializes and tests the media.

Logical and Physical Disk Organization

All disks used by OS-9 use the first few sectors to store basic identification, file structure, and storage allocation information.

LSN 0 is the “identification sector.” LSN 1 is the “disk allocation map sector.” LSN 2 marks the beginning of the disk’s root directory. The following section tells more about LSN 0 and LSN 1.

Disk Sectors

Identification Sector (LSN 0)

LSN 0 contains a description of the physical and logical characteristics of the disk. These characteristics are set by the Format command program when the disk is initialized.

The table below gives the OS-9 mnemonic name, byte address, size, and description of each value stored in this LSN 0.

Name	Relative Address	Size (Bytes)	Use
DD.TOT	\$00	3	Number of sectors on disk
DD.TKS	\$03	1	Track size (in sectors)
DD.MAP	\$04	2	Number of bytes in the allocation bit map
DD.BIT	\$06	2	Number of sectors per cluster
DD.DIR	\$08	3	Starting sector of the root directory
DD.OWN	\$0B	2	Owner’s user number
DD.ATT	\$0D	1	Disk attributes
DD.DSK	\$0E	2	Disk identification (for internal use)
DD.FMT	\$10	1	Disk format, density, number of sides
DD.SPT	\$11	2	Number of sectors per track
DD.RES	\$13	2	Reserved for future use
DD.BT	\$15	3	Starting sector of the bootstrap file
DD.BSZ	\$18	2	Size of the bootstrap file (in bytes)
DD.DAT	\$1A	5	Time of creation: Y:M:D:H:M
DD.NAM	\$1F	32	Volume name: last character has the most significant bit set

Disk Allocation Map Sector (LSN 1)

LSN 1 contains the “disk allocation map,” which is created by Format. This map specifies which sector clusters are available for file allocation.

The DD.MAP value in LSN 0 specifies the number of bytes (in LSN 1) that are used in the map.

Each bit on the map corresponds to one sector cluster on the disk. The DD.BIT value in LSN 0 specifies the number of sectors per cluster. The number is an integral power of 2 (1, 2, 4, 8, 16, and so on). The map may contain up to 4096 bits. Therefore, hard disks and double-density, doubled-sided diskettes have two or more sectors per cluster.

If a cluster is available, the corresponding bit is cleared. If it is allocated, non-existent, or physically defective, the corresponding bit is set.

File Sectors

File Descriptor Sector

The first sector of every file is the “file descriptor.” It contains the logical and physical description of the file.

The table below describes the contents of the file descriptor.

Name	Relative Address	Size (Bytes)	Use
FD.ATT	\$00	1	File attributes: D S PE PW PR E W R (see below)
FD.OWN	\$01	2	Owner's user ID
FD.DAT	\$03	5	Date last modified: Y M D H M
FS.LNK	\$08	1	Link count
FD.SIZ	\$09	4	File size (number of bytes)
FD.DCR	\$0D	3	Date created: Y M D
FD.SEG	\$10	240	Segment list (see below)

FD.ATT (the attribute byte) contains the file permission bits. When set the bits indicate the following:

Bit 7	Directory
Bit 6	Sharable
Bit 5	Public execute
Bit 4	Public write
Bit 3	Public read
Bit 2	Execute
Bit 1	Write
Bit 0	Read

FD.SEG (the segment list) consists of up to 48 5-byte entries that have the size and address of each file block in logical order. Each entry has the block's 3-byte LSN and 2-byte size (in sectors). The entry following the last segment will be zero.

After creation, a file has no data segments allocated to it until the first write. (Write operations past the current end-of-file cause sectors to be added to the file. The first write is always past the end-of-file.)

If the file has no segments, it is given an initial segment. Usually, this segment has the number of sectors specified by the minimum allocation entry in the device descriptor. If, however, the number of sectors requested is more than the minimum, the initial segment has the requested number.

Later expansions of the file usually are made in minimum allocation increments, also. OS-9 expands the last segment, whenever possible, instead of adding a segment. When the file is closed, unused sectors in the last segment are truncated.

Note: OS-9 tries to minimize the number of storage segments used in a file. In fact, many files have only one segment. In such cases, no extra read operations are needed to randomly access any byte on the file.

If a file is repeatedly closed, opened, and expanded, it may become fragmented and may have many segments. You can avoid this problem by writing a byte at the highest address to be used on a file. Do this before writing any other data.

Directories

“Disk directories” are files that have the D attribute set. A directory contains an integral number of entries, each of which can hold the name and LSN of a file or another directory.

Each directory entry contains 29 bytes for the filename, followed by the 3-byte LSN of the file’s descriptor sector. The filename is left-justified in the field with the sign bit of the last character set. Unused entries have a zero byte in the first filename character position.

Every disk has a master directory called the “root directory.” The DD.DIR value in LSN 0 (identification sector) specifies the starting sector of the root directory.

RBFMAN Definitions of the Path Descriptor

As stated earlier in this chapter, the PD.FST section of the path descriptor is reserved for and defined by the file manager. The table below describes the use of this section by RBFMAN. For your convenience, it also includes the other sections of the PD.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers and device drivers)			
PD.PD	\$00	1	Path number
PD.MOD	\$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	\$02	1	Number of open images (paths using this PD)
PD.DEV	\$03	2	Address of the associated device table entry
PD.CPR	\$05	1	Current process ID
PD.RGS	\$06	2	Address of the caller’s 6809 register stack
PD.BUF	\$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
RBFMAN Path Descriptor Definitions (PD.FST Section)			
PD.SMF	\$0A	1	State flag: 0 = current buffer is altered, 1 = current sector is in the buffer, 2 = descriptor sector is in the buffer
PD.CP	\$0B	4	Current logical file position (byte address)
PD.SIZ	\$0F	4	File size
PD.SBL	\$13	3	Segment beginning logical sector number (LSN)
PD.SBP	\$16	3	Segment beginning physical sector number (PSN)
PD.SSZ	\$19	2	Segment size
PD.DSK	\$1B	2	Disk ID (for internal use only)
PD.DTB	\$1D	2	Address of drive table
RBFMAN Option Section Definitions (PD.OPT Section)			
(Copied from the device descriptor)			
	\$20	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
PD.DRV	\$21	1	Drive number (0 . . n)
PD.STP	\$22	1	Step rate
PD.TYP	\$23	1	Device type
PD.DNS	\$24	1	Density capability
PD.CYL	\$25	2	Number of cylinders (tracks)
PD.SID	\$27	1	Number of sides (surfaces)
PD.VFY	\$28	1	0 = verify disk writes
PD.SCT	\$29	2	Default number of sectors per track
PD.T0S	\$2B	2	Default number of sectors per track (Track 0)
PD.ILV	\$2D	1	Sector interleave factor
PD.SAS	\$2E	1	Segment allocation size
(Not copied from the device descriptor)			
PD.ATT	\$33	1	File attributes (D S P E P W P R E W R)
PD.FD	\$34	3	File descriptor PSN
PD.DFD	\$37	3	Directory file descriptor PSN
PD.DCP	\$3A	4	File's directory entry pointer
PS.DVT	\$3E	2	Address of the device table entry

Any values not determined by this table default to zero.

RBF-Type Device Descriptor Modules

This section describes the use of the initialization table contained in the device descriptor modules for RBF-type devices. The values below are those IOMAN copies from the device descriptor to the path descriptor.

STP#0 = C11F
STP#1 = C14E

Name	Relative Address	Size (Bytes)	Use
	\$0-\$11		Standard device descriptor module header
IT.DTP	\$12	1	Device type: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
IT.DRV	\$13	1	Drive number (see below)
IT.STP	\$14	1	Step rate (see below)
IT.TYP	\$15	1	Device type (see below)
IT.DNS	\$16	1	Media density: 0 = single, 1 = double (see below)
IT.CYL	\$17	2	Number of cylinders (tracks)
IT.SID	\$19	1	Number of sides
IT.VFY	\$1A	1	0 = Verify disk writes
IT.SCT	\$1B	2	Default number of sectors per track
IT.TOS	\$1D	2	Default number of sectors per track (Track 0)
IT.ILV	\$1F	1	Sector interleave factor
IT.SAS	\$20	1	Minimum size of segment allocation (number of sectors to be allocated at one time)

IT.DRV is used to associate a 1-byte integer with each drive that a controller handles. The drives for each controller should be numbered 0 to $n-1$, where n is the maximum number of drives the controller can handle.

IT.STP (floppy disks) sets the head stepping rate to be used with a drive. To reduce access time, this should be the fastest rate the drive can support. The actual values stored depend on the specific disk controller and disk driver module used. Below are the values that are used by the popular Western Digital Floppy Disk Controller IC (FD179X 5-Inch):

<u>Step Code</u>	<u>Rate</u>
0	30ms
1	20ms
2	12ms
3	6 ms

IT.TYP specifies the device type (all types).

- Bit 0 — 0 = 5-inch floppy disk
 1 = 8-inch floppy disk
- Bit 5 — 0 = Non-Color Computer format
 1 = Color Computer format
- Bit 6 — 0 = Standard OS-9 format
 1 = Non-standard format
- Bit 7 — 0 = Floppy disk
 1 = Hard disk

IT.DNS specifies the density capabilities (floppy disk only).

- Bit 0 — 0 = Single-bit density (FM)
 1 = Double-bit density (MFM)
- Bit 1 — 0 = Single-track density (5-inch, 48
 tracks per inch)
 1 = Double-track density (5-inch, 96
 tracks per inch)

RBF-Type Device Driver Modules

An RBF-type device driver module contains a package of sub-routines that perform sector-oriented I/O to or from a specific hardware controller. These modules are usually reentrant. Because of this, one copy of the module can simultaneously run several devices that use identical I/O controllers.

IOMAN allocates a permanent memory area for each device (which may control several drives). The size of the memory area is given in the device driver module header. IOMAN and RBFMAN use some of this area. The device driver may use the rest in any manner. This area is used as follows:

RBF Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use
V.PAGE	\$00	1	Port extended address (A20 -A16) 24-bit device address
V.PORT	\$01	2	Device base address (defined by IOMAN)
V.LPRC	\$03	1	ID of the last active process (Not used by RBF device drivers)
V.BUSY	\$04	1	ID of the active process; 0 = not busy (defined by RBFMAN)
V.WAKE	\$05	1	ID of the process to reawaken after device completes I/O; 0 = no process waiting (defined by device driver)
V.USER		.	End of the OS-9 definitions
V.NDRV	\$06	1	Number of drives the controller can use (defined by the device driver)
DRVBEG		.	Beginning of the drive tables
TABLES	\$07	DRVMEN*N	Number of tables reserved (<i>n</i>)
FREE			Free for the driver to use

V.PAGE through V.USER are pre-defined in the OS-9Defs file. V.NDRV, DRVBEG, DRVMEN are pre-defined in the RBFDefs file.

TABLES. This area contains one table for each drive that the controller will handle. (RBFMAN assumes that there are as many tables as indicated by V.NDRV). Some time after the driver Init routine has been called, RBFMAN issues a request

for the driver to read LSN 0 (the identification sector) from a drive table by copying the first part of LSN 0 (up to DD.SIZ) into it.

The format of each drive table is as given below:

Name	Relative Address	Size (Bytes)	Use
DD.TOT	\$00	3	Number of sectors.
DD.TKS	\$03	1	Track size (in sectors).
DD.MAP	\$04	2	Number of bytes in the allocation bit map.
DD.BIT	\$06	2	Number of sectors per bit (cluster size).
DD.DIR	\$08	3	Address (LSN) of the root directory.
DD.OWN	\$0B	2	Owner's user number.
DD.ATT	\$0D	1	Disk access attributes: D S P E P W P R E W R.
DD.DSK	\$0E	2	Disk ID (a pseudo-random number used to detect disk swaps).
DD.FMT	\$10	1	Media format.
DD.SPT	\$11	2	Number of sectors per track. (Track 0 may use a different value, specified by IT.TOS in the device descriptor.)
DD.RES	\$15	2	Reserved for future use.
DD.SIZ		.	
V.TRAK	\$15	2	Number of the current track (the track the head is on — the one updated by the driver).
V.BMB	\$17	1	Bit-map use flag: 0 = the bit map is not in use. (Disk driver routines must not alter V.BMB.)
DRVMEN	\$18	.	Size of each drive table.

The format attributes (DD.FMT) are these:

Bit B0 = Number of sides

0 = Single-sided

1 = Double-sided

Bit B1 = Density

0 = Single-density

1 = Double-density

Bit B2 = Track density

0 = Single (48 tracks per inch)

91 = Double (96 tracks per inch)

RBFMAN Device Driver Subroutines

Like all device driver modules, RBFMAN device drivers use a standard executable memory module format. In the header, the type is specified as code \$0E (device driver).

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a long branch (LBRA) to the corresponding subroutine. The branch table is defined as follows:

ENTRY	LBRA	INIT	Initialize drive
	LBRA	READ	Read sector
	LBRA	WRITE	Write sector
	LBRA	GETSTA	Get status
	LBRA	SETSTA	Set status
	LBRA	TERM	Terminate device

Each subroutine should exist with the C bit of the Condition Code Register cleared if no error occurred. If an error occurred, the C bit should be set and an appropriate error code returned in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init

Function:

Initialize the device and its memory area.

Entry Conditions:

U = *address of the device memory area*
Y = *address of the device descriptor*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Technical Function:

1. If the disk writes are verified, use the Request Memory system call (F\$srqmem) to allocate a 256-byte buffer area where a sector may be read back and verified after a write.
2. Initialize the device memory area. For floppy disk controllers this typically consists of:
 - Initializing V.NDRV to the number of drives with which the controller will work
 - Initializing DD.TOT (in the drive table) to a non-zero value so that Sector 0 may be read or written.
 - Initializing V.TRAK to \$FF so that the first seek finds Track 0.
3. Place the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F\$irq).
4. Initialize the device control registers (enable interrupts if necessary).

Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These contain the 24-bit device address.) The driver should initialize each drive table appropriately for the type of disk the driver expects to use on the corresponding drive.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it isn't clear, the driver goes back to sleep.

Function:

Terminates a device. This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).

Read

Function:

Read a 256-byte sector from the disk and place it in the 256-byte buffer.

Entry Conditions:

U	=	<i>address of the device memory area</i>
Y	=	<i>address of the path descriptor</i>
B	=	<i>MSB of the disk LSN</i>
X	=	<i>LSBs of the disk LSN</i>

Exit Conditions:

The sector is returned in the sector buffer.

If error:

CC	=	C bit set
B	=	<i>error code</i>

Technical Function:

1. Get the sector buffer address from PD.BUF in the path descriptor.
2. Get the drive number from PD.DRV in the path descriptor.
3. Compute the physical disk address from the logical sector number.
4. Initiate the read operation.
5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it isn't clear, the driver goes back to sleep.

Whenever LSN 0 is read, the first part of this sector must be copied into the proper drive table. (Get the drive number from PD.DRV in the path descriptor.) The number of bytes to copy is in DD.SIZ.

The drive number (PD.DRV) should be used to compute the offset to the corresponding drive table as follows:

LDA PD.DRV,Y	Get the drive number
LDB #DRVMEN	Get the size of a drive table
MUL	
LEAX DRVBEG,U	Get the address of the first table
LEAX D,X	Compute the address of Table <i>n</i>

Write

Function:

Write a 256-byte sector buffer to the disk.

Entry Conditions:

U	= <i>address of the device memory area</i>
Y	= <i>address of the path descriptor</i>
B	= <i>MSB of the disk LSN</i>
X	= <i>LSBs of the disk LSN</i>

Exit Conditions:

The sector buffer is written out to the disk.

If error:

CC	= <i>C bit set</i>
B	= <i>error code</i>

Technical Function:

1. Get the sector buffer address from PD.BUF in the path descriptor.
2. Get the drive number from PD.DRV in the path descriptor.
3. Compute the physical disk address from the logical sector number.
4. Initiate the write operation.
5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete. (The IRQ service routine is responsible for sending the wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it is not, the driver goes back to sleep. If the disk controller cannot be interrupt-driven, it is necessary to perform a programmed I/O transfer.

-
6. If PF.VFY in the path descriptor is equal to zero, read the sector back in and verify that it was written correctly. This usually does not involve a comparison of the data.

If disk writes are to be verified, the Init routine must request the buffer where the sector may be placed when it is read back in. Do not copy LSN 0 into the drive table when it is read back to be verified.

Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the Read routine.

Getsta Setsta

Function:

Get/set the device's operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

U	=	<i>address of the device memory area</i>
Y	=	<i>address of the path descriptor</i>
A	=	<i>status code</i>

Exit Conditions:

Depend upon the function code

If error:

CC	=	C bit set
B	=	<i>error code</i>

It may be necessary to examine or change the register stack that contains the values of the 6809 register at the time of the call. The address of the register stack is in PD.RGS, which is located in the path descriptor.

The following offsets may be used to access any value in the register stack:

Name	Relative Address	Size (Bytes)	6809 Register
R\$CC	\$00	1	Condition Code Register
R\$D	\$01	.	Register D
R\$A	\$01	1	Register A
R\$B	\$02	1	Register B
R\$DP	\$03	1	Register DP
R\$X	\$04	2	Register X
R\$Y	\$06	2	Register Y
R\$U	\$08	2	Register U
R\$PC	\$0A	2	Program Counter

Term

Function:

Terminate a device. This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).

Entry Conditions:

U = address of the device memory area

Exit Conditions:

None

If error:

CC = C bit set
B = error code

Technical Function:

1. Wait until any pending I/O is completed.
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.
4. If the Init routine reserved a 256-byte buffer for verifying disk writes, return the memory with the Return Sysmem system call (F\$srtmem).

IRQ Service Routine

This routine is not included in the device driver's branch table and is not called directly by RBFMAN, but it is a key routine in interrupt-driven device drivers.

The IRQ service routine services interrupts and, when I/O is complete, sends a wakeup signal to the process whose process ID is in V.WAKE. It also clears V.WAKE as a flag to the mainline program that the IRQ has indeed occurred.

When it finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.

Technical Function:

1. Service the device interrupts (receive data from device or send data to it). This routine should put its data into and get its data from buffers which are defined in the device memory area.
2. Wake up any process that is waiting for I/O to complete. To do this, it checks to see if there is a process ID in V.WAKE (non-zero); if so, it sends a wakeup signal to that process.
3. If the device is ready to send more data and the output buffer is empty, disable the device's "ready to transmit" interrupts.

-
4. If a pause character is received, set V.PAUS in the attached device storage area to a non-zero value. The address of the attached device memory area is in V.DEV2.

Boot (Bootstrap Module)

Function:

Load the Boot file into memory from mass-storage.

Entry Conditions:

None

Exit Conditions:

D = *size of the boot file (in bytes)*
X = *address where the boot file was loaded into memory*

If error:

CC = *C bit set*
B = *error code*

The Boot module is **not** part of the disk driver. It is a separate module which is normally co-resident with the OS9P2 module in the system firmware.

The bootstrap module contains one subroutine that loads the bootstrap file and some related information into memory. It uses the standard executable module format with a module type of “system” (code \$C). The execution offset in the module header contains the offset to the entry point of this subroutine.

It obtains the starting sector number and size of the OS9Boot file from LSN 0. OS-9 is called to allocate a memory area large enough for the Boot file, and then it loads the Boot file into this memory area.

Technical Function:

1. Read LSN 0 from the disk into a buffer area. Boot must pick its own buffer area. LSN 0 contains the values for DD.BT (the 24-bit LSN of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes).
2. Get the 24-bit LSN of the bootstrap file from DD.BT.
3. Get the size of the bootstrap file from DD.BSZ. The Boot is contained in one logically contiguous block beginning at the logical sector specified in DD.BT and extending for $(DD.BSZ/256 + 1)$ sectors.
4. Use the OS-9 Request Sysmem system call (F\$srqmem) to request the memory area where the Boot file will be loaded.
5. Read the Boot file into this memory area.
6. Return the size of the Boot file and its location.

6 / Sequential Character File Manager

The Sequential Character File Manager (SCFMAN) supports devices that operate on a character-by-character basis. These include terminals, printers, and modems.

It is a reentrant subroutine package called by IOMAN for I/O system calls to sequential, character-oriented devices. It includes the extensive I/O editing functions typical of line-oriented operation (backspace, line delete, repeat line, auto line feed, screen pause, and return delay padding).

OS-9 is supplied with SCFMAN and one SCR-type device driver module. The module is an RS-232-type, which runs the serial interface.

SCFMAN Line Editing Functions

Read and Write

The Read and Write system calls (I\$read and I\$write) to SCFMAN-type devices correspond to the BASIC09 GET and PUT statements. They pass data to/from the device with little modification.

Note: Although there is otherwise little modification, the keyboard interrupt, keyboard abort, and pause character are filtered out of the input. (Editing is disabled if the corresponding character in the path descriptor contains a zero.)

Carriage returns are **not** followed by line feeds or nulls automatically, and the high order bits are passed as sent/received.

Read Line and Write Line

The Read Line and Write Line system calls (I\$readln and I\$writln) to SCFMAN devices correspond to the BASIC09 INPUT, PRINT, READ, and WRITE statements. They perform full line editing of all functions enabled for the particular device.

These functions are initialized when the device is first used. (The option table is copied from the device descriptor table associated with the specific device.)

Later, they may be altered — either from assembly-language programs using the Get Status system call, or from the keyboard using the Tmode command. All bytes transferred in this mode will have the high order bit cleared.

SCFMAN Definitions of the Path Descriptor

As you know, the PD.FST and PD.OPT sections of the path descriptor are reserved for and used by the file manager.

The table below describes the use of PD.FST and PD.OPT by SCFMAN. For your convenience, it also includes the other sections of the PD.

The PD.OPT section contains the values that determine the line editing functions. It contains many device operating parameters which may be read or written by the Set Status or Get Status system call. Any values not set by this table default to zero.

Note: It is possible to disable most of the editing functions by setting the corresponding control character in the path descriptor to zero. You can use the Set Status system call or the Tmode command to do this. Or, you may go a step further by setting the corresponding control character value in the device descriptor module to zero.

To determine the default settings for specific devices, you may inspect the device descriptors.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers)			
PD.PD	\$00	1	Path number
PD.MOD	\$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	\$02	1	Number of open images (paths using this PD)
PD.DEV	\$03	2	Address of the associated device table entry
PD.CPR	\$05	1	Current process ID

PD.RGS	\$06	2	Address of the caller's 6809 register stack
PD.BUF	\$08	2	Address of the 256-byte data buffer (if used)

SCFMAN Path Descriptor Definitions (PD.FST Section)

PD.DV2	\$0A	2	Device table address of the second (echo)device
PD.RAW	\$0C	1	Edit flag: 0 = raw mode, 1 = edit mode
PD.MAX	\$0D	2	Read Line maximum character count
PD.MIN	\$0F	1	Devices are "mine" if cleared
PD.STS	\$10	2	Status routine module address
PD.STM	\$12	2	Reserved for status routine

Name	Relative Address	Size (Bytes)	Use
------	------------------	--------------	-----

SCFMAN Option Section Definition (PD.OPT Section)

(Copied from the device descriptor)

	\$20	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF
PD.UPC	\$21	1	Case: 0 = upper and lower, 1 = upper only
PD.BSO	\$22	1	Backspace: 0 = backspace, 1 = backspace then space and backspace
PD.DLO	\$23	1	Delete: 0 = backspace over line, 1 = carriage return/line feed
PD.EKO	\$24	1	Echo: 0 = no echo
PD.ALF	\$25	1	Auto line feed after carriage return: 0 = no auto line feed
PD.NUL	\$26	1	End-of-line null count: 1 = nulls (\$00) sent after each carriage return/line feed
PD.PAU	\$27	1	Number of lines (since last input) before each end-of-page pause: 0 = no pause
PD.PAG	\$28	1	Lines per page
PD.BSP	\$29	1	Backspace character
PD.DEL	\$2A	1	Delete-line character
PD.EOR	\$2B	1	End-of-record character (end-of-line character) read only: Normally set to \$0D; 0 = Terminate Read Line only at the end of the file

PD.EOF	\$2C	1	End-of-file character (read only)
PD.RPR	\$2D	1	Reprint-line character
Name	Relative Address	Size (Bytes)	Use
PD.DUP	\$2E	1	Duplicate-last-line character
PD.PSC	\$2F	1	Pause character
PD.INT	\$30	1	Keyboard-interrupt character CLEAR C
PD.QUT	\$31	1	Keyboard-abort character CLEAR Q
PD.BSE	\$32	1	Backspace-echo character
PD.OVF	\$33	1	Line-overflow character (bell)
PD.PAR	\$34	1	Device-initialization value (parity)
PD.BAU	\$35	1	Software settable baud rate
PD.D2P	\$36	2	Offset to second device name string
PD.STN	\$38	2	Offset of status routine name
PD.ERR	\$3A	1	Most recent I/O error status

PD.EOF specifies the end-of-file character. If this is the first and only character input, SCFMAN returns an end-of-file error on Read or Readln.

PD.PSC specifies the pause character, which suspends output before the next end-of-record character. The pause character also deletes any type-ahead input for Readln.

PD.INT specifies the keyboard-interrupt character. When the character is received on input, a keyboard interrupt signal is sent to the last user of this path. The character also terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code.

PD.QUT specifies the keyboard-abort character. When this character is received on input, a keyboard abort signal is sent to the last user of this path. It also terminates the current I/O request (if any) with an error code identical to the keyboard

SCF-Type Device Descriptor Modules

The chart below shows the use of the initialization table in the device descriptors for SCF-type devices. The values are those IOMAN copies from the device descriptor to the path descriptor.

An SCF editing function is turned off if its corresponding value is set to zero. For example, if IT.EOF is set to zero, there is no end-of-file character.

Name	Relative Address	Size (Bytes)	Use
TABLE	.		Beginning of the option table
IT.DVC	\$12	1	Device class: 0 = SCF, 1 = RBF, 2 = PIPE, 3 = SCF
IT.UPC	\$13	1	Case: 0 = upper and lower case, 1 = upper only
IT.BSO	\$14	1	Backspace: 0 = backspace; 1 = backspace then space and backspace
IT.DLO	\$15	1	Delete: 0 = backspace over line, 1 = carriage return
IT.EKO	\$16	1	Echo: 0 = no echo
IT.ALF	\$17	1	Auto line feed: 0 = no auto line feed
Name	Relative Address	Size (Bytes)	Use
IT.NUL	\$18	1	End-of-line null count
IT.PAU	\$19	1	Pause: 0 = no end-of-page pause
IT.PAG	\$1A	1	Lines per page
IT.BSP	\$1B	1	Backspace character
IT.DEL	\$1C	1	Delete-line character
IT.EOR	\$1D	1	End-of-record character
IT.EOF	\$1E	1	End-of-file character
IT.RPR	\$1F	1	Reprint-line character
IT.DUP	\$20	1	Duplicate-last-line character
IT.PSC	\$21	1	Pause character
IT.INT	\$22	1	Interrupt character
IT.QUT	\$23	1	Quit character
IT.BSE	\$24	1	Backspace echo character

IT.OVF	\$25	1	Line-overflow character (bell)
IT.PAR	\$26	1	Initialization value (parity) used to initialize the device's control register when a path is opened to it
IT.BAU	\$27	1	Baud rate
IT.D2P	\$28	2	Attached device name string offset
IT.STN	\$2A	2	Offset to status routine
IT.ERR	\$2C	1	Initial error status

SCF-Type Device Driver Modules

A SCFMAN-type device driver module contains a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. For each “incarnation” of the driver, IOMAN allocates a permanent memory area for that device.

The size of the memory area is given in the device driver module header. IOMAN and SCFMAN use some of this area. The device driver may use the rest in any way (typically as variables and buffers). The area is used as follows:

SFC Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use
V.PAGE	\$00	1	Port extended address
V.PORT	\$01	2	Device base address
V.LPRC	\$03	1	ID of the last active process
V.BUSY	\$04	1	ID of the active process: 0 = not busy (defined by RBFMAN)
V.WAKE	\$05	1	ID of the process to reawaken after the device completes I/O: 0 = no process is waiting (defined by the device driver)
V.USER		.	End of the OS-9 definitions
V.TYPE	\$06	1	Device type of parity
V.LINE	\$07	1	Lines left until the end of the page
V.PAUS	\$08	1	Pause request: 0 = no pause
V.DEV2	\$09	2	Attached device memory area
V.INTR	\$0B	1	Interrupt character
V.QUIT	\$0C	1	Quit character
V.PCHR	\$0D	1	Pause character
V.ERR	\$0E	1	Error accumulator
V.SCF	\$0F	1	End of the SCFMAN definitions
FREE		.	Free for the device driver to use

V.LPRC contains the process-ID of the last process to use the device. The IRQ service routine is responsible for sending this process the proper signal in case a quit character or an interrupt character is received. V.LPRC is defined by SCFMAN.

V.BUSY contains the process ID of the process that is using the device. (If the device is not being used, V.BUSY contains a zero.) This is used by SCFMAN to prevent more than one process from using the device at the same time. V.BUSY is defined by SCFMAN.

SCFMAN Device Driver Subroutines

Like all device drivers, SCFMAN device drivers use a standard executable memory module format. In the header the type is specified as code \$0E (device driver).

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is defined as follows:

ENTRY	LBRA	INIT	Initialize drive
	LBRA	READ	Read sector
	LBRA	WRITE	Write sector
	LBRA	GETSTA	Get status
	LBRA	SETSTA	Set status
	LBRA	TERM	Terminate device

Each subroutine should exist with the C bit in the Condition Code Register cleared if no error occurred. If an error occurred, the C bit should be set and an appropriate error code returned in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init

Function:

Initialize the device control registers (enable interrupts, if necessary).

Entry Conditions:

U = *address of the device memory area*
Y = *address of the device descriptor*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Technical Function:

1. Initialize the device memory area.
2. Place the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F\$irq).
3. Initialize the device control registers.

Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These contain the device address.) There is no need to initialize the part of the memory area used by IOMAN and SCFMAN.

Read

Function:

Read the next character from the input buffer.

Entry Conditions:

U = *address of the device memory area*

Y = *address of the path descriptor*

Exit Conditions:

A = *character read*

If error:

CC = C bit set

B = *error code*

Technical Function:

1. Get the next character from the input buffer.
2. If no data is ready, Read copies its process ID from V.BUSY into V.WAKE. It then uses the Sleep system call to put itself to sleep.

Later, when data is received, the Set IRQ system call leaves the data in a buffer, then checks V.WAKE to see if any process is waiting for the device to complete I/O. If so, Set IRQ should send a wakeup signal to it.

Data buffers are **not** allocated automatically. If a buffer is used, it should be defined in the device memory area.

Write

Function:

Output a character (place a data byte into an output buffer) and enable the device output interrupts.

Entry Conditions:

U = *address of the device memory area*
Y = *address of the path descriptor*
A = *character to write*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

1. If the data buffer is already full, Write copies its process ID from V.BUSY into V.WAKE. It then puts itself to sleep.

Getsta Setsta

2. Later, when the IRQ service routine transmits a character and makes room for more data, Write checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, it sends a wakeup signal to that process.

Write must ensure that the IRQ service routine will start up when data is placed into the buffer. After an interrupt is generated, the IRQ service routine continues to transmit data until the data buffer is empty. Then, it disables the device's "ready to transmit" interrupts.

Data buffers are **not** automatically allocated. If a buffer is used, it should be defined in the device memory area.

Function:

Get/set the device's operating parameter (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

U	=	<i>address of the device memory area</i>
Y	=	<i>address of the path descriptor</i>
A	=	<i>status code</i>

Exit Conditions:

Depend upon function code

Currently, all of the function codes defined by Microware for SCF-type devices are handled by IOMAN or SCFMAN. Any codes not defined by Microware are passed to the device driver.

It may be necessary to examine or change the register stack that contains the values of the 6809 registers at the time of the call. The address of the register stack may be found in PD.RGS, which is located in the path descriptor.

The following offsets may be used to access any value in the register packet:

Name	Relative Address	Size (Bytes)	6809 Register
R\$CC	\$00	1	Conditions Code Register
R\$D	\$01	.	Register D
R\$A	\$01	1	Register A
R\$B	\$02	1	Register B
R\$DP	\$03	1	Register DP
R\$X	\$04	2	Register X
R\$Y	\$06	2	Register Y
R\$U	\$08	2	Register U
R\$PC	\$0A	2	Program Counter

Term

Function:

Terminate a device. This routine is called when a device is no longer in use (when the link count of its device descriptor module becomes zero).

Entry Conditions:

U = *pointer to the device memory area*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Technical Function:

1. Wait until the output buffer is emptied (by the IRQ service routine).
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.

Note: Permanent storage used by device drivers is never returned to the free memory pool. Therefore, you should **not** terminate any device that might be used again. Modules contained in the Boot file will **never** be terminated.

IRQ Service Routine

This routine is not included in the device driver's branch table and is not called directly by SCFMAN, but it is a key routine in device drivers.

The IRQ service routine services device interrupts, and when I/O is complete, it sends a wakeup signal to the process whose process ID is in V.WAKE. It also clears V.WAKE as a flag to the mainline program that the IRQ has occurred.

When it finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.

For technical information, see "IRQ Service Routine" in *Chapter 5*.

7 / Assembly-Language Programming Techniques

There are four key rules to follow when you write OS-9 assembly-language programs:

- The program must use position-independent code (PIC). OS-9 selects program load addresses based on available memory at runtime. You cannot select them.
- The program must use the standard OS-9 memory module format; otherwise, it cannot be loaded and run. It cannot use self-modifying code. It must not change anything in a memory module or use any part of the module for variables.
- Storage for all variables and data structures must be within the data area OS-9 assigns at runtime. This area is separate from the program memory module.
- All input and output operations should be made using OS-9 system calls.

The 6809's versatile addressing modes make it easy for you to follow these rules. The OS-9 assembler also helps; it has special capabilities to assist you in creating programs and memory modules for the OS-9 execution environment.

How to Write Position-Independent Code

The 6809 instruction set allows efficient use of position-independent code.

The basic technique is to always use PC-relative addressing — for example, BRA, LBRA, BSR and LBSR. Get addresses of constants and tables using LEA instructions instead of load immediate instructions. If you use dispatch tables, use tables of relative addresses, not absolute addresses.

Incorrect	Correct
LDX #CONSTANT	LEAX CONSTANT,PCR
JSR SUBR	BSR SUBR or LBSR SUBR
JMP LABEL	BRA LABEL or LBRA LABEL

Addressing Variables and Data Structures

A program that is executed as a process (by the Fork and Chain system calls or by the shell) is assigned part of RAM for variables, stacks, and data structures at execution time. The RAM area cannot be specified or determined ahead of time. However, a minimum size for the area is specified in the program's module header.

When the program is first entered, Register Y contains the address of the top bound of the process's data memory area. Register U contains the address of the lower bound, and Register DP contains the area's page number.

The creating process may have passed a parameter area. If so, this parameter area is located from the value of X and the SP to the top of memory (Register Y), and Register D contains the area's size in bytes.

If the new process was called by the shell, the parameter area contains the part of the shell command line that includes the argument (parameter) text. (I/O redirection arguments are not included.)

The most important rule is this: **Do not use extended addressing.** You should use only indexed and direct page addressing to access data area values and structures. Do not use program-counter relative addressing to find addresses in the data area, but do use it to refer to addresses within the program area.

The most efficient way to handle tables, buffers, and stacks is to have the program's initialization routine compute their absolute addresses. It does this, using the data area bounds passed by OS-9 in the registers.

The absolute addresses then can be saved in the direct page where they can be loaded into registers quickly, using short instructions. Direct page addressing has these advantages: it is faster than extended addressing, and the program is inherently reentrant.

Stack Requirements

OS-9 uses interrupts extensively, and many reentrant 6809 programs use the 6809 register stack for local variable storage. Because of this, a generous stack should be maintained at all times. We recommend no fewer than 200 bytes.

Interrupt Masks

User programs should keep the F and I bits (FIRQ mask and IRQ mask) of the Condition Codes Register off. To avoid task-switching or interrupts during critical program sequences, you can set these bits. You should set them for no longer than a tick, however. Otherwise, system time-keeping may no longer be accurate.

Using Standard I/O Paths

Write your programs to use standard I/O paths wherever practical. Usually, this involves I/O calls that are intended to communicate to the user's terminal, or any other case where the OS-9 redirected I/O capability is desirable.

All three standard I/O paths are open when the program is entered; they are inherited from the parent process. Programs should **not** close these paths except under special circumstances.

The standard I/O paths are assigned as follows:

- Path 0 — Standard input. Analogous to the keyboard or other main data input source.
- Path 1 — Standard output. Analogous to the terminal display or other main data output destination.
- Path 2 — Standard error/status. This path is provided so output messages that are not part of the actual program output can be kept separate. Paths 1 and 2 are often directed to the same device.

Writing Interrupt-Driven Device Drivers

OS-9 programs do not use interrupts directly. Any interrupt-driven function should be implemented as a device driver module which should handle all interrupt-related functions. When a program must be synchronized to an interrupt-causing event, a driver can send a semaphore to a program (or vice versa), using OS-9's signal facilities.

It is important to understand that interrupt service routines are asynchronous and are not distinct processes. They are, in effect, subroutines called by OS-9 when an interrupt occurs.

Therefore, all interrupt-driven device drivers have two basic parts: the "mainline" subroutines that execute as part of the calling process, and a separate interrupt service routine. The two routines are asynchronous and therefore must use signals for communications and coordination.

The Init initialization subroutine within the driver package should do the following:

1. Allocate memory area for the service routine
2. Get the service routine address
3. Execute the Set IRQ system call to add it to the IRQ polling table

When a device driver routine does something that results in an interrupt, it should immediately execute a Sleep system call. This turns off the process.

When the interrupt occurs, its service routine is executed after some random interval. It should then do the least amount of processing required, and send a wakeup signal to its associated process using the Send system call. It may also put some data in its memory area (I/O data and status), which is shared with its associated sleeping process.

Later, the signal awakens the device driver mainline routine. The routine can now process the data or status returned by the interrupt service routine.

The OS-9 List utility command program is shown on this and the next page as an example of assembly-language programming.

Microware OS-9 Assembler RS Version 01.00.00 06-17/83 13:06:11 Page 001
LIST — OS-9 System Symbol Definitions

```

00001      *****
00002      *
00003      * LIST UTILITY COMMAND
00004      * Syntax: list <Pathname>
00005      * COPIES INPUT FROM SPECIFIED FILE TO STANDARD OUTPUT
00006      *
00007      NAM LIST
00008      * NOTE: A USE /D0/DEFS/OS9DEFS
00009      * IS ENCLOSED WITHIN THE IFP1/ENOC STATEMENT
00010      IFP1
00012      ENOC
00013      0011      TYPE      SET  PRGRM+OBJECT
00014      0081      REVS      SET  REENT+1
00015      0000 87CD004E      MOD  LSTEND,LSTNAM,PRGRM+OBJECT,
                                REENT+1,LSTENT,

00016      * NOTE THE 'MOD' CMD CONSISTS OF:
00017      * LSTEND,LSTNAM,PRGRM+OBJECT,REENT+1,LSTENT,LSTMEM
00018      000D 4C4953D4 LSTNAM FCS "LIST"
00019      *
00020      * STATIC STORAGE OFFSETS
00021      *
00022      00C8      BUFSIZ    equ    200      size of input buffer
00023      D 0000      org      0
00024      D 0000      IPATH    rmb      1      input path number
00025      D 0001      PRMPTR    rmb      2      parameter pointer
00026      D 0003      BUFFER    rmb    BUFSIZ    allocate line buffer
00027      D 00CB      rmb      200      allocate stack
00028      D 0193      rmb      200      room for parameter
                                list
00029      D 025B      LSTMEM    EQU      .
00030      0011 9F01      LSTENT    stx      PRMPTR    save parameter ptr
00031      0013 8601      lda      #read    select read access
                                mode
00032      0015 103F84      os9      I$open    open input file
00033      0018 252E      bcs      LIST50    exit if error
00034      001A 9700      sta      IPATH    save input path
                                number
00035      001C 9F01      stx      PRMPTR    save updated param
                                ptr
00036      001E 9600      LIST20    lda      IPATH    load input path
                                number
00037      0020 3043      leax      BUFFER,U    load buffer ptr
00038      0022 108E00CB      ldy      #BUFSIZ    maximum bytes to be
                                read

```

00039	0026	103F8B		os9	I\$readln	read line of input
00040	0029	2509		bcs	LIST30	exit if error
00041	002B	8601		lda	#1	load std. out path
00042	002D	103F8C		os9	I\$writln	output the line
00043	0030	24EC		bcc	LIST20	repeat if no error
00044	0032	2014		bra	LIST50	exit if error
00045	0034	C1D3	LIST30	cmpb	#E\$EOF	at end of file?
00046	0036	2610		bne	LIST50	branch if not
00047	0038	9600		lda	IPATH	load input path number
00048	003A	103F8F		os9	I\$close	close input path
00049	003D	2509		bcs	LIST50	..exit if error
00050	003F	9E01		ldx	PRMPTR	restore Param Ptr
00051	0041	A684		lda	0,X	
00052	0043	810D		cmpa	##0D	end of parameter line?
00053	0045	26CA		bne	LSTENT	..no; list next file
00054	0047	5F		clrb		
00055	0048	103F06	LIST50	os9	F\$exit	...terminate
00056	004B	BF7690		emod		module crc
00057	004E		LSTEND	equ	*	
00058				end		

00000 error(s)
00000 warning(s)

Microware OS-9 Assembler RS Version 01.00.00 06-17/83 13:11:28 Page 002
LIST — OS-9 System Symbol Definitions

\$004E 00078 program bytes generated
\$025B 00603 data bytes allocated
\$1CAE bytes used for symbols

8 / System Calls

System calls are used to communicate between the OS-9 operating system and assembly-language programs. There are two major types of calls — I/O calls and function calls. Function calls include user mode calls and system mode calls.

Each system call has a mnemonic name. Names of I/O calls start with I\$. For example, the Change Directory call is I\$chgdir. Names of function calls start with F\$. For example, the Allocate Bits call is F\$allbit. The names are defined in the assembler-entry conditions equate file called OS9Defs.

The I/O calls are:

Attach	Make Directory
Change Directory	Open Path
Close Path	Read
Create File	Read Line
Delete File	Seek
Detach Device	Set Status
Duplicate Path	Write
Get Status	Write Line

The user calls are:

Allocate Bits	Parse Name
Chain	Print Error
Compare Names	Search Bits
CRC	Send
Deallocate Bits	Set Priority
Exit	Set SVC
Fork	Set SWI
Get ID	Set Time
Intercept	Sleep
Link	Time
Load	Unlink
Memory	Wait

The system mode calls are:

Allocate 64	Request Sysmem
Find 64	Return 64
I/O Delete	Return Sysmem
I/O Queue	Set IRQ
Insert Process	Verify Module
Next Process	

System mode calls are privileged. They may be executed only while OS-9 is in the system state (when it is processing another system call, executing a file manager or device driver, and so on).

System mode calls are included in this manual primarily for programmers who will be writing device drivers and other system-level applications. In the “System Call Descriptions” section of this chapter, system mode calls are listed separately from I/O and user calls.

Appendices A and B, the system call quick reference lists, summarize each system call’s function.

Calling Procedure

All system calls are executed via an **SWI2** instruction.

1. Load the 6809 register with any appropriate parameters.
2. Execute a SWI2 instruction, followed immediately by a constant byte, which is the request code.
3. After OS-9 processes the call, it returns any parameters in the 6809 register. If an error occurred, the C bit of the Condition Code Register is set, and Accumulator B contains the appropriate error code. This permits a BCS or BCC instruction immediately following the system call to branch on error/no error.

As an example, here is the Close system call:

```
LDA    PATHNUM
SWI2
FCB    $8B
BCS    ERROR
```

You can use the assembler's "OS9" directive to simplify the call. Here is the simplified Close system call:

```
LDA    PATHNUM
OS9    I$CLOSE
BCS    ERROR
```

OS-9 accepts system calls in **any combination of upper- or lower-case letters.**

I/O System Calls

OS-9's I/O calls are easier to use than many other systems' I/O calls. This is because the calling program does not have to allocate and set up "file control blocks", "sector buffers," and so on.

Instead, OS-9 returns a 1-byte path number when a path to a file/device is opened or created. Until the path is closed, this path number may be used in later I/O requests to identify the file or device.

In addition, OS-9 allocates and maintains its own data structures. You need not deal with them.

System Call Descriptions

The rest of this chapter consists of the system call descriptions. At the top of each description is the system call name, followed by its mnemonic and code. Next is a summary of the call's function, entry conditions, and exit conditions. When further explanation is required, it is included under a section called "Technical Function."

In the system call descriptions, registers not specified as entry or exit conditions are not altered. Strings passed as parameters are normally terminated by having Bit 7 of the last character set, a space character, or an end-of-line character.

If an error occurs on a system call, the C bit of Register CC is set, and Register B contains the *error code*. If no error occurs, the C bit is clear, and Register B contains a value of zero.

Allocate Bits

OS9 F\$allbit

103F 13

Function:

Sets bits in the allocation bit map specified by Register X.

Bit numbers range from 0 to $n-1$, where n is the number of bits in the allocation bit map.

Entry Conditions:

X = *starting address of the allocation bit map*
D = *number of the first bit to set*
Y = *number of bits to set*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Function:

Attaches a new device to the system or verifies that a device is attached.

Attach does not “reserve” the device. It only prepares it for later use by any process.

Most devices are installed automatically. Therefore, you need to use Attach only when installing a device dynamically or when verifying the existence of a device. You need not use the Attach system call to perform routine I/O.

The *access mode* parameter specifies the read and/or write operations to be allowed. These are:

- 0 = Use device capabilities
- 1 = Read
- 2 = Write
- 3 = Update

Entry Conditions:

- X = *address of the device name string*
- A = *access mode*

Exit Conditions:

- U = *address of the device table entry*

If error:

- CC = C bit set
- B = *error code*

Technical Function:

1. OS-9 searches the system module to see if memory contains a device descriptor that has the same name as the device.
- 2a. If it finds the descriptor and if the device is not already attached, OS-9 links to its file manager and device driver. It then places their addresses in a new device table entry. OS-9 then allocates any memory needed by the device driver and calls the driver's initialization routine (which usually initializes the hardware).
- 2b. If it finds the descriptor and if the device is already attached, OS-9 does not reinitialize the device.

Function:

Loads and executes a new primary module, but does not create a new process. A Chain system call is similar to a Fork followed by an Exit, but with less processing overhead.

Chain “resets” the calling process’s program and data memory areas and begins executing a new primary module. It does not affect open paths.

Warning: The hardware stack pointer (Register SP) should be located somewhere in the direct page before the Chain is executed. This helps prevent a “suicide” (system crash) or a “suicide attempt” error. It also prevents a suicide in the event that the new module requires a smaller data area than that in use. You should allow approximately 200 bytes of stack space for execution of the Chain system call.

Entry Conditions:

X	= <i>address of the module name or filename</i>
Y	= <i>parameter area size (in pages); defaults to zero if not specified</i>
U	= <i>starting address of the parameter area</i>
A	= <i>language/type code</i>
B	= <i>size of the data area (in pages); must be at least one page</i>

Exit Conditions:

None

If error:

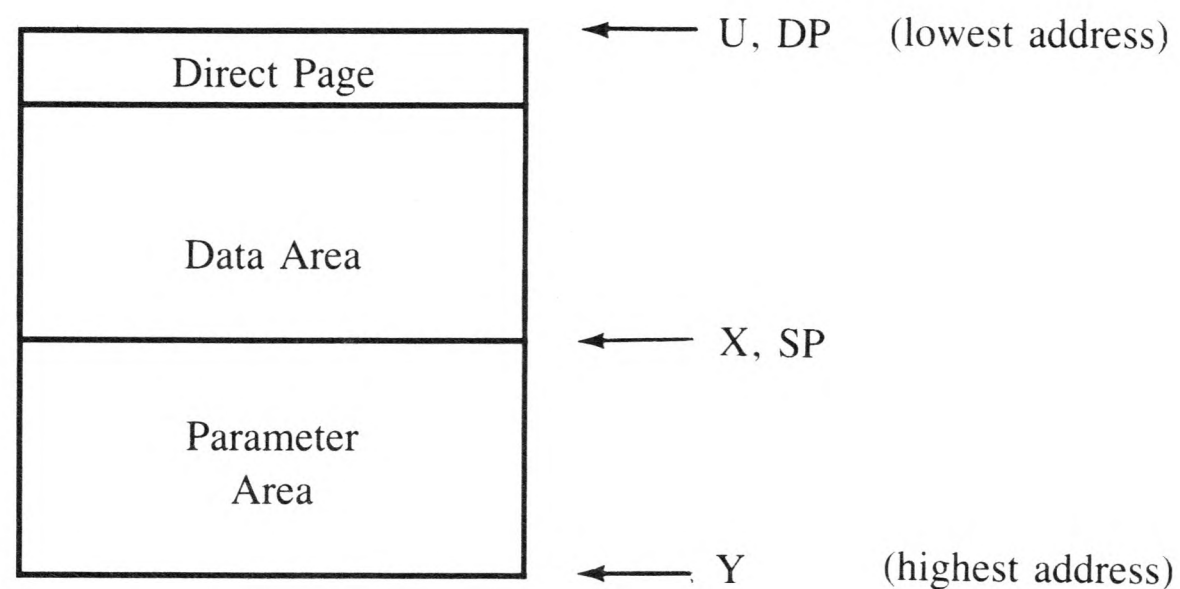
CC	= C bit set
B	= <i>error code</i>

Technical Function:

1. OS-9 passes the name string of the new process’s primary module (the program that is to be executed first). Then, it searches the system module directory to see if a module with the same name, type, and language is already in memory.

- 2a. If the module is in memory, it is linked to.
- 2b. If the module is not in memory, the name string is used as the pathlist of a file which is to be loaded into memory. Then, the first module in this file is "linked to." (Several modules may have been loaded from a single file.)
3. OS-9 unlinks the process's old primary module.
4. The data memory area is reconfigured to the size specified in the new primary module's header.

The diagram below shows how Chain sets up the data memory area and registers for the new module.



D = *parameter area size*
 PC = *module entry point absolute address*
 CC = *F=0, I=0; others are undefined*

Registers Y and U (the top-of-memory and bottom-of-memory pointers, respectively) always have values at page boundaries. If the parent process does not specify a size for the parameter area, the size (Register D) defaults to zero. The data area must be at least one page long.

(For more information, see the Fork system call.)

Change Directory

OS9 I\$chgdir

103F 86

Function:

Changes a process's working directory to the directory specified by the pathlist.

If the access mode is read, write, or update, the current data directory is changed. If the access mode is execute, the current execution directory is changed.

The calling process must have read access to the directory specified (public read if the directory is not owned by the calling process).

The *access modes* are:

- 1 = Read
- 2 = Write
- 3 = Update
- 4 = Execute

Entry Conditions:

- X = *address of the pathlist*
- A = *access mode*

Exit Conditions:

None

If error:

- CC = C bit set
- B = *error code*

Close Path

OS9 I\$close

103F 8F

Function:

Terminates the I/O path to the file or device specified by the *path number*. Unless you use another Open or Create system call, you can no longer perform I/O to the file or device.

Non-sharable devices become available to other requesting processes. All OS-9 internally managed buffers and descriptors are deallocated.

The Exit system call automatically closes all open paths (except the standard I/O paths). Therefore, you may not need to use the Close Path system call to close them.

Do not close standard I/O paths unless you want to change the files or devices to which they correspond.

Entry Conditions:

A = *path number*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Compare Names

OS9 F\$cmpnam

103F 11

Function:

Compares two strings and indicates whether they match. Use this call with the Parse Names system call.

The second name must have the most significant bit (Bit 7) of the last character set.

Entry Conditions:

X = *address of string1*
B = *length of string1*
Y = *address of string2*

Exit Conditions:

CC = C bit clear if the strings match

If error:

CC = C bit set
B = *error code*

Function:

Calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators.

The calculation begins at the *starting byte address* and continues over the specified *number of bytes*.

You need not cover an entire module in one call, since the CRC may be “accumulated” over several calls. The CRC accumulator can be any 3-byte memory area and must be initialized to \$FFFFFF before the first CRC call.

The last three bytes in the module are not included in the CRC generation. The three CRC bytes are to be stored here.

Entry Conditions:

X	=	<i>starting byte address</i>
Y	=	<i>number of bytes</i>
D	=	<i>address of the 3-byte CRC accumulator</i>

Exit Conditions:

The CRC accumulator is updated.

If error:

None

Create File

OS9 I\$create

103F 83

Function:

Creates and opens a file on a disk.

OS-9 parses the pathlist and enters the new filename in the specified directory — or the working directory, if none is specified.

The file is given the attributes passed in Register B, which has bits defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Sharable file

The *access mode* parameter passed in Register A must be either write or update. This mode affects the file only until it is closed. The file can be reopened in any access mode allowed by the file attributes (see the Open system call).

Files opened for write may allow faster data transfer than those opened for update because update sometimes needs to pre-read sectors. These access codes are defined as follows: Bit 2 = write; Bit 3 = update.

Note: If the execute bit (Bit 2) is set, the file is created in the working execution directory instead of the working data directory.

The path number returned by OS-9 is used to identify the file in later I/O system calls until the file is closed.

No data storage is initially allocated for the file at the time it is created. This is done automatically by the Write subroutine or explicitly by the Setsta subroutine.

Deallocate Bits

OS9 F\$delbit

103F 14

Function:

Clears bits in the allocation bit map pointed to by Register X.

Bit numbers range from 0 to $n-1$, where n is the number of bits in the allocation bit map.

Entry Conditions:

X = *starting address of the allocation bit map*
D = *number of the first bit to set*
Y = *number of bits to set*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Delete File

OS9 I\$delete

103F 87

Function:

Deletes the disk file specified by the pathlist.

The file must have write permission attributes (public write if the calling process is not the owner).

An attempt to delete a device results in an error.

Entry Conditions:

X = *address of the pathlist* (see example below)

Exit Conditions:

X = *address of the last byte of the pathlist + 1*;
any trailing blanks are skipped (see example below)

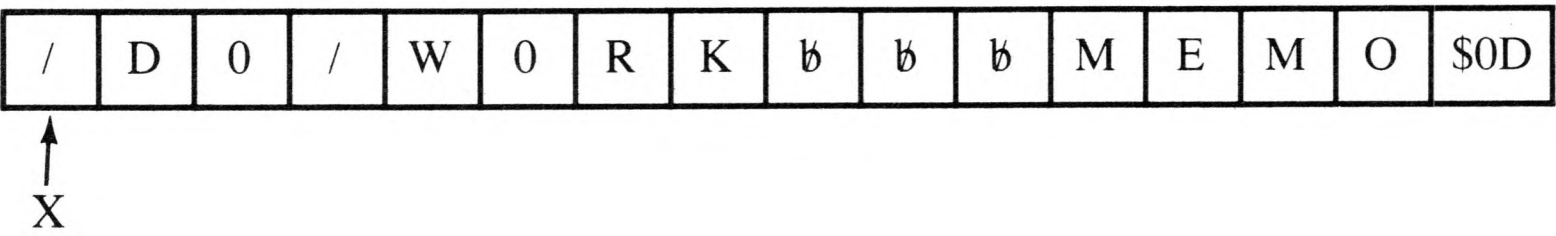
If error:

CC = C bit set

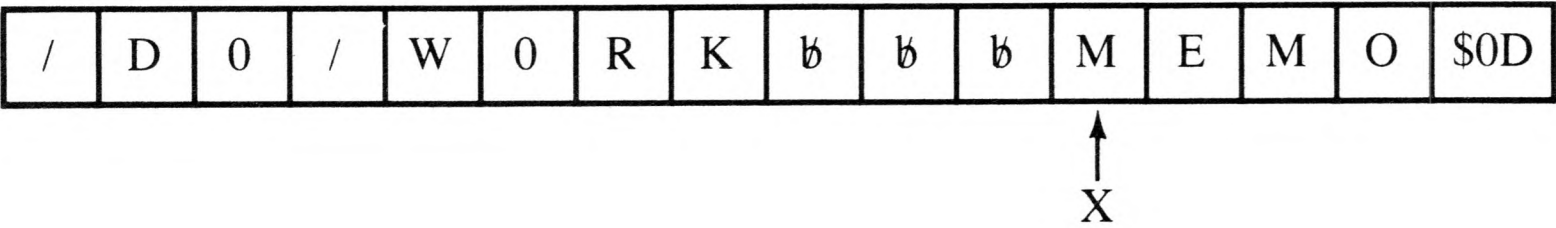
B = *error code*

Example:

Before the Delete File call;



After the Delete File call;



Detach Device

OS9 I\$detach

103F 81

Function:

Removes a device from the system and the system device table, if the device is not being used by another other process.

You must use this call to detach devices that were attached using the Attach system call. Both calls are used mainly by IOMAN. SCFMAN also uses Attach/Detach to set up its second device (echo device).

Entry Conditions:

U = *address of the device table entry*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Technical Function:

1. The device driver's termination routine is called, then OS-9 deallocates any memory assigned to the driver.
2. The associated device driver and file manager modules are unlinked. If not being used by any other module, the driver is removed from RAM.

Duplicate Path

OS9 I\$dup

103F 82

Function:

Returns another, synonymous path number for the file or device specified by the *old path number*.

The shell uses this system call when it redirects I/O. System calls that use either path number (old or new) operate on the same file or device.

Entry Conditions:

A = *old path number* (number of path to duplicate)

Exit Conditions:

B = *new path number*

If error:

CC = C bit set

B = *error code*

Exit

OS9 F\$exit

103F 06

Function:

Terminates the calling process.

The Exit system call is the only way a process can “kill” itself. Exit deallocates the process’s data memory area and unlinks its primary module. It also closes all open paths automatically.

The Wait system call always returns to the parent the status code passed by the child in its Exit call. Therefore, if the parent executes a Wait and receives the status code, it knows the child has “died.”

Entry Conditions:

B = *status code to return to the parent*

Exit Conditions:

The process is terminated.

Fork

OS9 F\$fork

103F 03

Function:

Creates a new process, a “child” of the calling process. Fork also sets up the child process’s memory and 6809 register.

Entry Conditions:

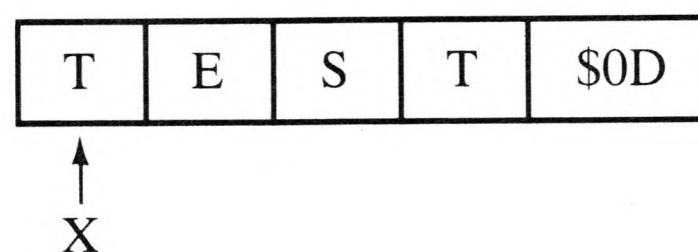
- X = *address of the module name or filename* (see example below)
- Y = *size of the parameter area* (in pages); defaults to zero if not specified
- U = *starting address of the parameter area*
- A = *language/type code*
- B = *size of the optional data area* (in pages); must be at least one page

Exit Conditions:

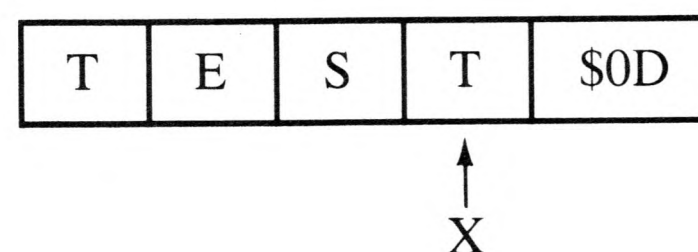
- X = *address of the last byte of the name + 1* (see example below)
- If error:
- CC = *C bit set*
 - B = *error code*

Example:

Before the Fork call:



After the Fork call:



Technical Function:

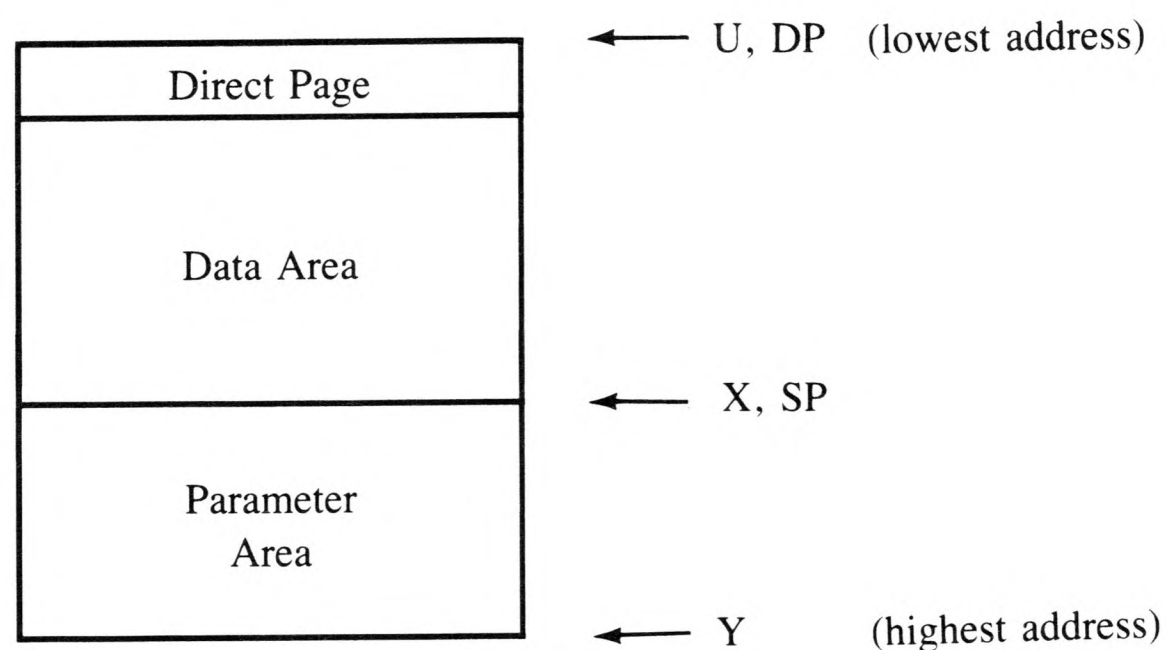
1. OS-9 passes the name string of the new process's "primary module" (the program that is to be executed first). Then, it searches the system module directory to see if the program already is in memory.
 - 2a. If the program is in memory, the module is linked to and executed.
 - 2b. If the program is not in memory, OS-9 uses the name as the pathlist of the file that is to be loaded into memory. Then, the first module in this file is linked to and executed. (Several modules may have been loaded from one file.)
3. OS-9 uses the primary module's header to determine the initial size of the process's data area. It then tries to allocate a contiguous RAM area of that size. (This area includes the parameter passing area, which is copied from the parent process's data area.)
4. The new process's registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point.

When the shell processes a command line, it passes a string in the parameter area. This string is a copy of the parameter part of the command line. To simplify string-oriented processing, it also inserts an end-of-line character at the end of the parameter string.

Register X points to the starting of the parameter string. If the command line included the optional memory size specification (*#n* or *#nK*), the shell passes that as the requested memory size when executing the Fork.

If any of the above operations is unsuccessful, the Fork is aborted and the caller is returned an error.

The diagram below shows how Fork sets up the data memory area and registers for a newly created process.



D = size of the parameter area
PC = module entry point absolute address
CC = F=0, I=0, others are undefined

Registers Y and U (the top-of-memory pointer and bottom-of-memory pointer, respectively) always have values at page boundaries.

As stated earlier under “Entry Conditions,” if the parent does not specify the size of the parameter area, the size defaults to zero. The minimum overall data area size is one page.

Note: The child and parent processes execute at the same time. If the parent executes a Wait system call immediately after the Fork, it waits until the child “dies” before it resumes execution.

Be careful when recursively calling a program that uses the Fork system call; another child may be created with each “incarnation.” This continues until the process table becomes full.

Get ID

OS9 F\$Id

103F 0C

Function:

Returns the caller's *process ID* and *user ID*.

The *process ID* is a byte value from 1 to 255. It is assigned by OS-9 and is unique to the process.

The *user ID* is an integer from 0 to 65535. It is defined in the system password file, and is used by the file security system and a few other functions. Several processes can have the same user ID.

Entry Conditions:

None

Exit Conditions:

A = *process ID*

Y = *user ID*

If error:

CC = C bit set

B = *error code*

Get Status

OS9 I\$gttstt

103F 8D

Function:

Returns the status of a file or device.

This is a “wildcard” call. It is used to handle device parameters that:

- Are not the same for all devices
- Are highly hardware-dependent
- Must be user-changeable

The exact operation of the Get Status system call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's parameters for such functions as backspace character and echo on/off. The Get Status call is commonly used with the Set Status call.

The Get Status function codes that are currently defined are listed in the “Uses of Get Status” section below.

Entry Conditions:

A = *path number*
X = *MS 16 bits of the desired file position*
U = *LS 16 bits of the desired file position*

Exit Conditions:

None

If error:

CC = *C bit set*
B = *error code*

Uses of Get Status

Function codes 7 through 127 are reserved for future use.

Codes 128 through 255 and their parameter-passing conventions are user-definable. (See the sections on writing device drivers). The function code and register stack are passed to the device driver.

The following function codes are defined: \$00, \$01, \$02, \$05, \$06, \$12, and \$ 13. The parameter-passing conventions for these function codes are listed below.

SS.OPT (Function code \$00): Reads the option section of the path descriptor, and copies it into the 32-byte area pointed to by Register X.

Use this to determine the current settings for editing functions, such as echo and auto line feed. For a complete description of the status packet, see the section on path descriptors.

Entry Conditions:

A = *path number*
B = \$00
X = *address to receive status packet*

Exit Conditions:

Status packet

If error:

CC = C bit set
B = *error code*

SS.RDY (Function code \$01): Tests for data available on SCFMAN-supported devices.

Entry Conditions:

A = *path number*
B = \$01

Exit Conditions:

If ready:

CC = C bit clear
B = \$00

If not ready:

CC = C bit set
B = \$F6 (E\$SRNDY)

If error:

CC = C bit set
B = *error code*

SS.SIZ (Function code \$02): Gets the current file size (RBFMAN-supported devices only).

Entry Conditions

A = *path number*
B = \$02

Exit Conditions

X = *ms 16 bits of the current file size*
U = *ls 16 bits of the current file size*

If error:

CC = C bit set
B = *error code*

SS.POS (Function code \$05): Gets the current file position (RBFMAN-supported devices only).

Entry Conditions:

A = *path number*
B = \$05

Exit Conditions:

X = *MS 16 bits of the current file position*
U = *LS 16 bits of the current file position*

If error:

CC = C bit set
B = *error code*

SS.EOF (Function code \$06): Tests for the end of the file (EOF).

Entry Conditions:

A = *path number*
B = \$06

Exit Conditions:

If not EOF:

CC = C bit clear
B = \$00

If EOF:

CC = C bit set
B = \$D3 (E\$EOF)

If error:

CC = C bit set

B = *error code*

SS.DSTAT (Function code \$12): Returns the display status.

Entry Conditions:

A = *path number*

B = \$12

Exit Conditions:

X = *address of the graphics display memory*

Y = *graphics cursor address; x = MSB,*
y = LSB

A = *color code of the pixel at the cursor address*

SS.JOY (Function code \$13): Returns the joystick values.

Entry Conditions:

A = *path number*

B = \$13

X = 0 (right joystick), or

X = 1 (left joystick)

Exit Conditions:

X = *selected joystick x value (0-63)*

Y = *selected joystick y value (0-63)*

A = \$FF (if the fire button is on), or

A = \$00 (if the fire button is off)

Function:

Tells OS-9 to set a signal intercept trap. Then, whenever the process receives a signal, the process's intercept routine is executed.

Once the signal trap is set, the intercept routine may be executed at any time because a signal may occur at any time.

The intercept routine should terminate with an RTI instruction.

Note: If a process has not used the Intercept system call to set a signal trap, the process aborts if it receives a signal.

Entry Conditions:

X = *address of the intercept routine*
U = *starting address of the routine's memory area*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Technical Function:

1. When the process receives a signal, OS-9 sets Registers U and B as follows:

U = *starting address of the intercept routine's memory area*
B = *signal code (process's termination status)*

Note: The value of Register DP may not be the same as it was when the Intercept call was made.

2. After setting the registers, OS-9 transfers execution to the intercept routine.

Function:

Links to a memory module that has the specified name, language, and type.

The module's "link count" is incremented whenever Link references its name. This keeps track of how many processes are using the module.

If the module requested is not sharable (not reentrant), only one process may link to it at a time.

Entry Conditions:

X = *address of the module name* (see example below)
 A = *type/language byte*

Exit Conditions:

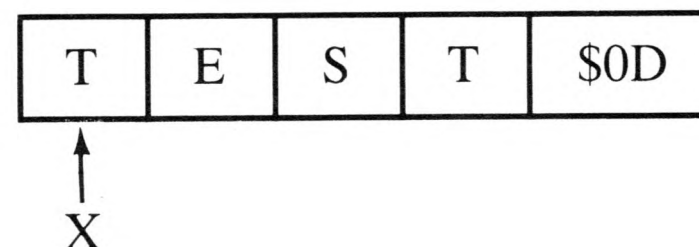
X = *address of the last byte of the module name + 1* (see example below)
 Y = *module entry point absolute address*
 U = *module header absolute address*
 A = *type/language code*
 B = *attributes / revision level*

If error:

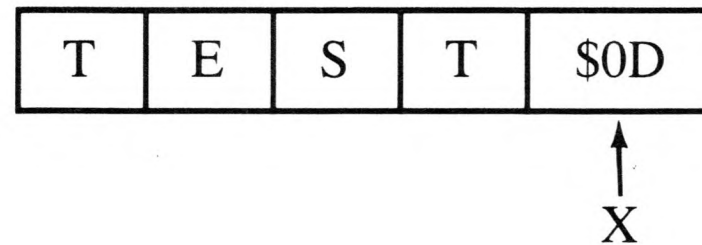
CC = *C bit set*
 B = *error code*

Example:

Before the Link call:



After the Link call:



Technical Function:

1. OS-9 searches the module directory for a module that has the specified name, language, and type.
- 2a. If OS-9 finds the module, the address of the module's header is returned in Register U, and the absolute address of the module's execution entry point is returned in Register Y. (This and other information can be obtained from the module header.)
- 2b. If OS-9 doesn't find the module — or if the type/language codes in the entry and exit conditions don't match — OS-9 returns an error.

Possible Errors:

- Module not found
- Module busy (not sharable and in use)
- Incorrect or defective module header

Function:

Loads a module or modules from the file specified by the pathlist or from the working execution directory (if no pathlist is given).

The file must have the execute access mode. It also must contain a module or modules that have a proper module header.

All modules loaded are added to the system module directory. The first module read is linked. The exit conditions apply only to the first module loaded.

Entry Conditions:

- X = *address of the pathlist (filename)* (see example below)
- A = *language/type code*; 0 = any language/type

Exit Conditions:

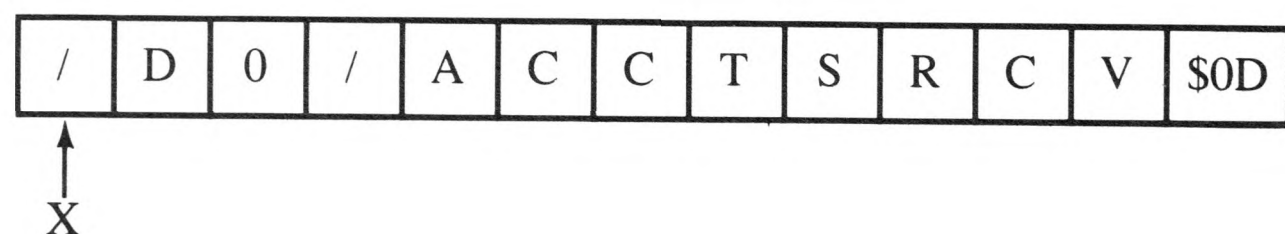
- X = *address of the last byte of the pathlist (filename) + 1* (see example below)
- Y = *primary module entry point address*
- U = *address of the module header*
- A = *language/type code*
- B = *attributes / revision level*

If error:

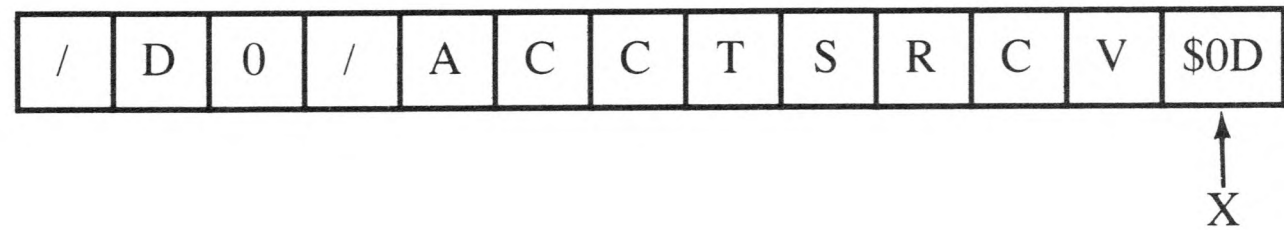
- CC = *C bit set*
- B = *error code*

Example:

Before the Load call:



After the Load call:



Possible errors:

- Module directory full
- Memory full
- Errors that occur on the Open, Read, Close, and Link system calls

Make Directory

OS9 I\$makdir

103F 85

Function:

Creates and initializes a directory as specified by the pathlist. The directory contains no entries, except for an entry for itself (.) and its parent directory (..)

The caller is made the owner of the directory. Because the Make Directory call does not open the directory, it does not return a path number.

The new directory automatically has its “directory” bit set in the access permission *attributes*. The remaining attributes are specified by the byte passed in Register B. The bits are defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Sharable
7	Don't care

Entry Conditions:

X = *address of the pathlist* (see example below)
B = *directory attributes*

Exit Conditions:

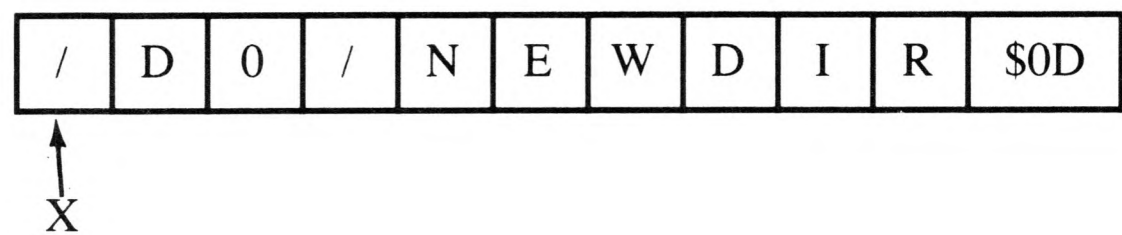
X = *address of the last byte of the pathlist + 1*;
any trailing blanks are skipped (see example below)

If error:

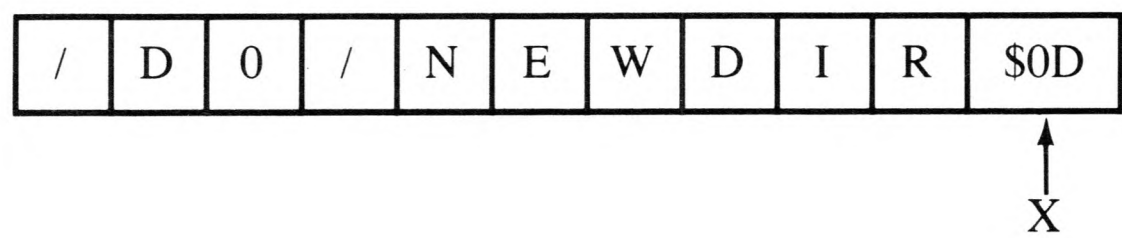
CC = C bit set
B = *error code*

Example:

Before the Make Directory call:



After the Make Directory call:



Function:

Expands or contracts the process's data memory area to the specified size. Or, if you specify zero as the new size, the call returns the current size of data memory.

The size requested is rounded off to the next page boundary. Additional memory is allocated contiguously upward or deallocated downward from the old highest address.

Even if enough free memory exists, the call may return an error upon an expansion request. This is because the data area is always made contiguous. Therefore, memory requests by other processes may fragment free memory into smaller, scattered blocks that are contiguous with the caller's present data area.

Entry Conditions:

D = *size of the new memory area (in bytes);*
 0 = return current size and upper bound

Exit Conditions:

Y = *address of the new memory area upper bound*
D = *actual size of the new memory (in bytes)*

If error:

CC = C bit set
B = *error code*

Open Path

OS9 I\$open

103F 84

Function:

Opens a path to an existing file or device as specified by the pathlist.

OS-9 searches for the file in one of the following:

- The directory specified by the pathlist if the pathlist begins with a slash
- The working data directory, if the pathlist does not begin with a slash
- The working execution directory, if the pathlist does not begin with a slash and if the execution bit is set in the access mode

A path number is returned; it is used in later system calls to identify the file.

The *access mode* parameter specifies which read and/or write operations are to be permitted. When set, the bits allow access as follows:

Bit(s)	Access
Read	Read
Write	Write
Read and write	Update
Directory	Directory I/O

The access mode must conform to the access permission attributes associated with the file or device (see the Create system call). Only the owner may access a file unless the appropriate “public permit” bits are set.

The update mode may be slightly slower than the others because pre-reading of sectors may be required for random access of bytes within sectors.

Several processes (users) may open files at the same time. Each device has an attribute that specifies whether or not it is sharable.

Entry Conditions:

X = *address of the pathlist* (see example below)
A = *access mode* (D S PE PW PR E W R)

Exit Conditions:

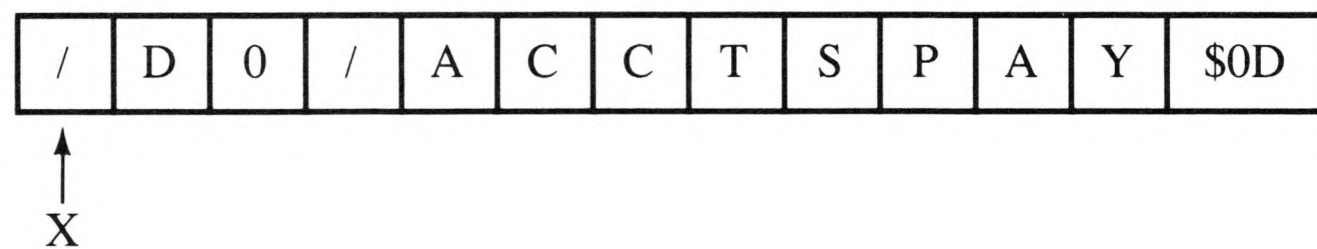
X = *address of the last byte of the pathlist + 1*
(see example below)
A = *path number*

If error:

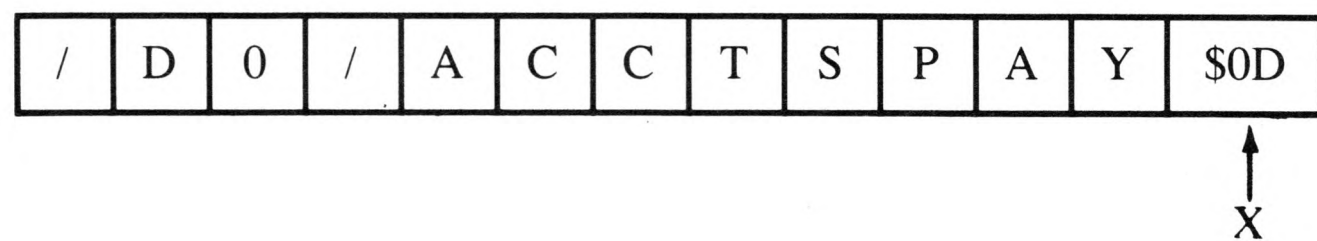
CC = C bit set
B = *error code*

Example:

Before the Open Path call:



After the Open Path call:



Parse Name

OS9 F\$prsnam

103F 10

Function:

Parses (analyzes) the input text string for a legal OS-9 name. The name is terminated by any character that is not a legal name character.

This system call is useful for processing pathlist arguments passed to new processes.

Because Parse Name processes only one name, several calls may be needed to process a pathlist that has more than one name. As you can see from the example below, Parse Name finishes with Register X in position for the next parse.

If Register X was at the end of a pathlist, a bad name error is returned. Then, Register X is moved past any space characters so that the next pathlist in a command line may be parsed.

Entry Conditions:

X = *address of the pathlist* (see example below)

Exit Conditions:

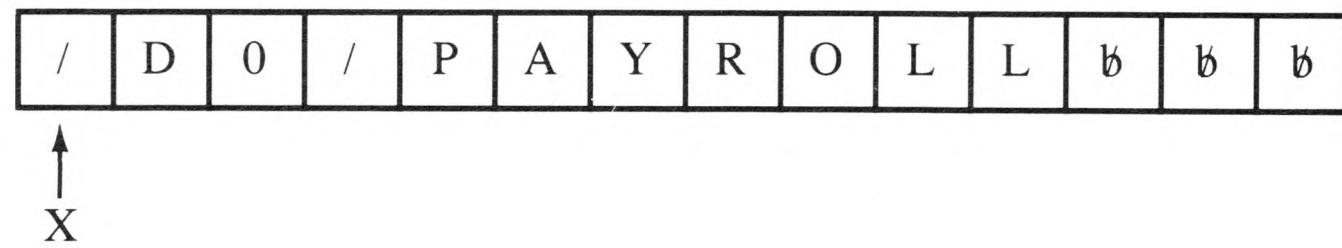
X = *address of the optional slash + 1*
Y = *address of the last character of the name*
 + 1 (see example below)
B = *length of the name*

If error:

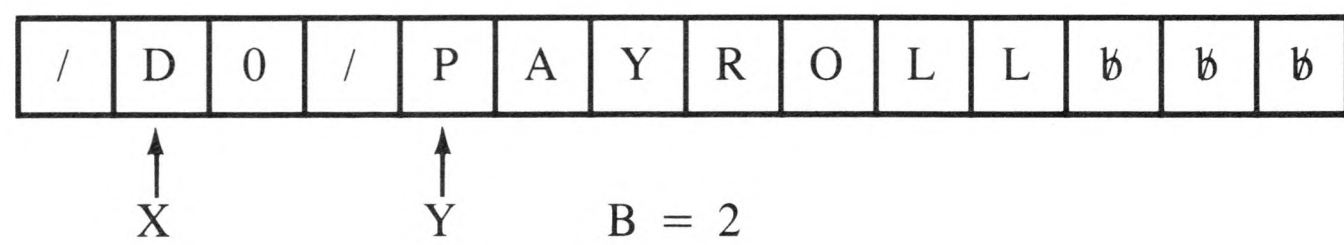
CC = C bit set
B = *error code*
X = *address of the last trailing blank + 1*

Example:

Before the Parse Name call:



After the Parse Name call:



Print Error

OS9 F\$perr

103F 0F

Function:

Writes an error message to the output path specified. By default, OS-9 displays:

ERROR #*decimal number*

The error reporting routine is vectored and can be replaced with a more elaborate reporting module. To replace this routine use the Set SVC system call.

Entry Conditions:

A = *output path number*
B = *error code*

Exit Conditions:

None
If error:
CC = C bit set
B = *error code*

Function:

Reads the specified number of bytes from the specified path. The data is returned exactly as read from the file/device without additional processing or editing.

The path must have been opened in the read or update mode.

After all data in a file is read, the next Read call returns an end-of-file error.

The keyboard abort, keyboard interrupt, and end-of-file characters may be filtered out of the entry conditions data on SCFMAN-type devices unless the corresponding entries in the path descriptor have been set to zero. You may want to modify the device descriptor so that these values are initialized to zero when the path is opened.

The number of bytes requested is read unless any of the following is true:

- An end-of-file occurs
- An end-of-record occurs (SCFMAN only)
- An error occurs

Entry Conditions:

X	= <i>address to store data</i>
Y	= <i>number of bytes to read</i>
A	= <i>path number</i>

Exit Conditions:

Y	= <i>number of bytes read</i>
---	-------------------------------

If error:

CC	= <i>C bit set</i>
B	= <i>error code</i>

Read Line

OS9 I\$readln

103F 8B

Function:

Reads a text line with editing.

Read Line is similar to Read except it reads the input file or device until a carriage return character is encountered or until the maximum byte count specified is reached. Line editing is also activated on character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.

SCFMAN requires that the last byte entered be an end-of-record character (usually a carriage return). If more data is entered than the maximum specified, it is not accepted and a PD.OVF character (usually bell) is echoed.

After all data in a file is read, the next Read Line call generates an end-of-file error.

(For more information about line editing, see “SCFMAN Line Editing Functions” in Chapter 6.)

Entry Conditions:

X	= <i>starting address to store data</i>
Y	= <i>maximum number of bytes to read</i>
A	= <i>path number</i>

Exit Conditions:

Y	= <i>number of bytes read</i>
---	-------------------------------

If error:

CC	= C bit set
B	= <i>error code</i>

Search Bits

OS9 F\$schbit

103F 12

Function:

Searches the specified allocation bit map for a free block (cleared bits) of the required length.

The search starts at the *starting bit number*. If no block of the specified size exists, the call returns with the carry set, starting bit number, and size of the largest block.

Entry Conditions:

X	=	<i>starting address of the map</i>
D	=	<i>starting bit number</i>
Y	=	<i>bit count (free bit block size)</i>
U	=	<i>ending address of the map</i>

Exit Conditions:

D	=	<i>starting bit number</i>
Y	=	<i>bit count</i>

Seek

OS9 I\$seek

103F 88

Function:

Repositions the path's logical "file pointer," the 32-bit address of the next byte in the file to be read from or written to.

A seek may be performed to any value, regardless of the file's size. Later writes automatically expand the file to the required size (if possible). Reads, however, return an end-of-file condition. Note that a seek to Address 0 is the same as a rewind operation.

Seeks to non-random access devices usually are ignored and return without error.

Entry Conditions:

A	= <i>path number</i>
X	= <i>MS 16 bits of the desired file position</i>
U	= <i>LS 16 bits of the desired file position</i>

Exit Conditions:

None

If error:

CC	= C bit set
B	= <i>error code</i>

Function:

Sends a signal to the specified process. The signal code is a single byte value from 1 through 255.

If the destination process is sleeping or waiting, it is activated so that it may process the signal.

If a signal trap was set up, the signal processing routine (Intercept) is executed (see the Intercept system call). If none was set up, the signal aborts the destination process, and the signal code becomes the exit status (see the Wait system call). An exception is the wakeup signal; it does not cause the signal intercept routine to be executed.

Signal codes are defined as follows:

0	= System abort (cannot be intercepted)
1	= Wake up the process
2	= Keyboard abort
3	= Keyboard interrupt
4-255	= User defined

If you try to send a signal to a process that has a signal pending, the current Send call is cancelled, and an error is returned. Issue a Sleep call for a few ticks, then try again. (The Sleep call saves CPU time.)

(See the Intercept, Wait, and Sleep system calls for more information.)

Entry Conditions:

A	= <i>destination's process ID</i>
B	= <i>signal code</i>

Exit Conditions:

None

If error:

CC	= C bit set
B	= <i>error code</i>

Set Priority

OS9 F\$sprior

103F 0D

Function:

Changes the process's priority to the *priority* specified.

A process can change another process's priority only if it has the same user ID.

Entry Conditions:

A = *process ID*
B = *priority*; 0 = lowest, 255 = highest

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Function:

Sets the status of a file or device.

This is a “wildcard” call. It is used to handle device parameters that:

- Are not the same for all devices
- Are highly hardware-dependent
- Must be user-changeable

The exact operation of the Set Status system call depends on the device driver and file manager associated with the path. A typical use is to set a terminal’s parameters for such functions as backspace character and echo on/off. The Set Status call is commonly used with the Get Status call.

The Set Status function codes that are currently defined are listed in the “Uses of Set Status” section below.

Entry Conditions:

A = *path number*

B = *function code*

Other registers depend upon the function code

Exit Conditions:

Depend upon the function code

If error:

CC = C bit set

B = *error code*

Uses of Set Status

Function codes 128 through 255 and their parameter-passing conventions are user-definable. (For more information, see the “RBF-Type Device Drivers” section in Chapter 5 and the “SCF-Type Device Drivers” section in Chapter 6.) The function code and register stack are passed to the device driver.

The following function codes are defined: \$00, \$02, \$04, \$09, \$0A, \$0B, \$0C, and \$0D. The parameter-passing conventions for these function codes are listed below.

SS.OPT (Function code \$00): Writes the option section of the path descriptor from the 32-byte status packet pointed to by Register X. Use this to set the device operating parameters, such as echo and line feed.

Entry Conditions:

A	=	<i>path number</i>
B	=	\$00
C	=	<i>address of the status packet</i>

Exit Conditions:

None

SS.SIZ (Function code \$02): Changes the file's size (RBFMAN-type devices only).

Entry Conditions:

A	=	<i>path number</i>
B	=	\$02
X	=	<i>MS 16 bits of the desired file size</i>
U	=	<i>LS 16 bits of the desired file size</i>

Exit Conditions:

None

SS.RST (Function code \$03): Restores the head to Track 0. This is used for formatting and error recovery.

Entry Conditions:

A	=	<i>path number</i>
B	=	\$03

Exit Conditions:

None

SS.WTK (Function code \$04): Formats (writes) a track on a floppy disk. For hard disks or floppy disks that have a “format entire disk command,” SS.WTK formats the entire disk only when the *track number* is zero.

Entry Conditions:

A = *path number*
B = \$04
X = *address of the track buffer*
U = *track number* (least significant 8 bits)
Y = *side/density*
Bit B0 = side (0 = Side 0, 1 = Side 1) Bit
B1 = density (0 = single, 1 = double)

Exit Conditions:

None

SS.FEE (Function code \$09): Issues a form feed.

Entry Conditions:

B = \$09

Exit Conditions:

None

SS.FRZ (Function code \$0A): Freezes the DD. information. (Inhibits the reading of LSN 0 to DDD.xxx variables, which define disk formats.) This enables the reading of non-standard disks.

Entry Conditions:

B = \$0A

Exit Conditions:

None

SS.SPT (Function code \$0B): Sets a different number of sectors per track so non-standard disks may be read.

Entry Conditions:

B = \$0B
X = *new sectors per track*

Exit Conditions:

None

SS.SQD (Function code \$0C): Starts the power-down sequence for hard disks that have sequence-down requirements prior to removal of power.

Entry Conditions:

B = \$0C

Exit Conditions:

None

SS.DCM (Function code \$0D): Transmits a command directly to a disk controller for specific functions. Parameters and commands are hardware-dependent for specific systems.

Entry Conditions:

B = \$0D

Other registers vary

Exit Conditions:

Vary

Set SVC

OS9 F\$ssvc

103F 32

Function:

Adds or replaces a system call, which you have written, in OS-9's user and system mode system call tables.

Entry Conditions:

Y = address of the system call initialization table

Exit Conditions:

None

If error:

CC = C bit set

B = error code

Technical Function:

1. Register Y passes the address of a table, which contains the function codes and offsets, to the corresponding system call handler routines. This table has the following format:

Use	
Relative Address	
\$00	Function Code ← First entry
\$01	Offset From Byte 3
\$02	To Function Handler
\$03	Function Code ← Second entry
\$04	Offset From Byte 6
\$05	To Function Handler
	More Entries ← Third entry . . .
	\$80 ← End-of-table mark

- 2a. If the most significant bit of the function code is set, the system table is updated.
- 2b. If the most significant bit of the function code is not set, the system and user tables are updated.

Function request codes are broken into these two categories:

\$00 - \$27	User mode system call codes.
\$29 - \$34	Privileged system mode system call codes. When these system calls are being installed, the most significant bit should be set if it is to be placed into the system table only.

These categories are defined by convention; they are not enforced by OS-9.

To use a privileged system call, you must be executing a program that has the type code \$0C (OS-9 system module).

The system call handler routine should process the system call and return from subroutine with an RTS instruction. The handler routine may alter all CPU registers (except Register SP). Register U passes the address of the register stack to the system call handler as shown in the following diagram:

	Relative Address	Name
U → CC	\$00	R\$CC
A	\$01	R\$D
B	\$01	R\$A
DP	\$02	R\$B
X	\$03	R\$DP
Y	\$04	R\$X
U	\$06	R\$Y
PC	\$08	R\$U
	\$0A	R\$PC

Codes \$25 through \$27 and \$70 through \$7F are user-definable.

Set SWI

OS9 F\$sswi

103F 0E

Function:

Sets the interrupt vectors for SWI2 and SWI3 instructions.

Each process has its own local vectors. Each Set SWI call sets one type of vector according to the code number passed in Register A:

- 1 = SWI
- 2 = SWI2
- 3 = SWI3

When a process is created, all three vectors are initialized with the address of the OS-9 service call processor.

Warning: Microware-supplied software uses SWI2 to call OS-9. If you reset this vector these programs cannot work. If you change all three vectors, you cannot call OS-9 at all.

Entry Conditions:

- A = *SWI type code*
- X = *address of the user software interrupt routine*

Exit Conditions:

None

If error:

- CC = *C bit set*
- B = *error code*

Set Time

OS9 F\$time

103F 16

Function:

Sets the current system date/time and starts the system real-time clock. The date and time are passed in a time packet as follows:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Entry Conditions:

X = *relative address of the time packet*

Exit Conditions:

The system time and date are set.

If error:

CC = C bit set
B = *error code*

Function:

Turns off the calling process temporarily.

If Register X contains 0, the process is turned off “indefinitely” (until it receives a signal). This is a good way to wait for a signal or interrupt without wasting CPU time.

If Register X contains 1, the process is turned off for the remainder of its current time slice. The process is inserted into the active process queue immediately, and resumes execution when it reaches the front of the queue. If the process receives a signal, it “awakens” before the time has elapsed.

If Register X contains an integer from 2 through 255, the process is turned off for the specified number of ticks, n . The process is inserted into the active process queue after $n-1$ ticks. It resumes execution when it reaches the front of the queue. If the process receives a signal, it awakens before the time has elapsed.

Entry Conditions:

X = *sleep time* (in ticks), or
X = 0 (sleep indefinitely), or
X = 1 (sleep for remainder of current time slice)

Exit Conditions:

X = *sleep time minus the number of ticks that the process was asleep*

If error:

CC = C bit set
B = *error code*

Function:

Returns the current system date and time in the form of a 6-byte packet (in binary). The packet is copied to the address passed in Register X.

The packet looks like this:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Entry Conditions:

X = *address of the area to store the time packet*

Exit Conditions:

time packet

If error:

CC = C bit set

B = *error code*

Function:

Unlinks (destroys) a module, if it is not being used by another process.

Device driver modules that are being used and certain system modules cannot be unlinked. ROM modules can be unlinked; they cannot, however, be deleted from the module directory.

Entry Conditions:

U = *address of the module header*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Technical Function:

1. Unlink tells OS-9 that the module is no longer needed by the calling process.
2. OS-9 decrements the module's link count.
- 3a. If the resulting link count is zero, OS-9 destroys the module.
- 3b. If any other process is using the module, the module's link count cannot decrement to zero. Therefore, OS-9 does not destroy the module.

Function:

Turns off the calling process until a child process “dies” by executing an Exit system call, or by receiving a signal. The Wait call helps you save system time.

The child’s process ID and exit status are returned to the parent. If the child died because of a signal, the exit status byte (Register B) is the signal code.

If the caller has several children, the caller is activated when the first one dies. Therefore, one wait system call is required to detect termination of each child.

If a child died before the Wait call, the caller is reactivated almost immediately. If the caller has no children, Wait returns an error.

(See the Exit system call for more information.)

Entry Conditions:

None

Exit Conditions:

A = *deceased child process’s ID*
B = *deceased child process’s exit status code*

If error:

CC = C bit set
B = *error code*

Write

OS9 I\$write

103F 8A

Function:

Writes to the file or device associated with the path number specified.

The path must have been opened or created in the write or update access mode. Data is written to the file or device without processing or editing. If data is written past the present end-of-file, the file is automatically expanded.

Entry Conditions:

X	= <i>starting address of data to write</i>
Y	= <i>number of bytes to write</i>
A	= <i>path number</i>

Exit Conditions:

Y	= <i>number of bytes written</i>
---	----------------------------------

If error:

CC	= <i>C bit set</i>
B	= <i>error code</i>

Write Line

OS9 I\$writln

103F 8C

Function:

Writes to the file or device associated with the path number specified.

Write Line is similar to Write except it writes data until a carriage return character is encountered. Line editing is also activated for character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.

The path must have been opened or created in the write or update access mode.

(For more information about line editing, see “SCFMAN Line Editing Functions” in Chapter 6.)

Entry Conditions:

X	= <i>starting address of the data to write</i>
Y	= <i>maximum number of bytes to write</i>
A	= <i>path number</i>

Exit Conditions:

Y	= <i>number of bytes written</i>
---	----------------------------------

If error:

CC	= <i>C bit set</i>
B	= <i>error code</i>

Allocate 64

OS9 F\$all64

103F 30

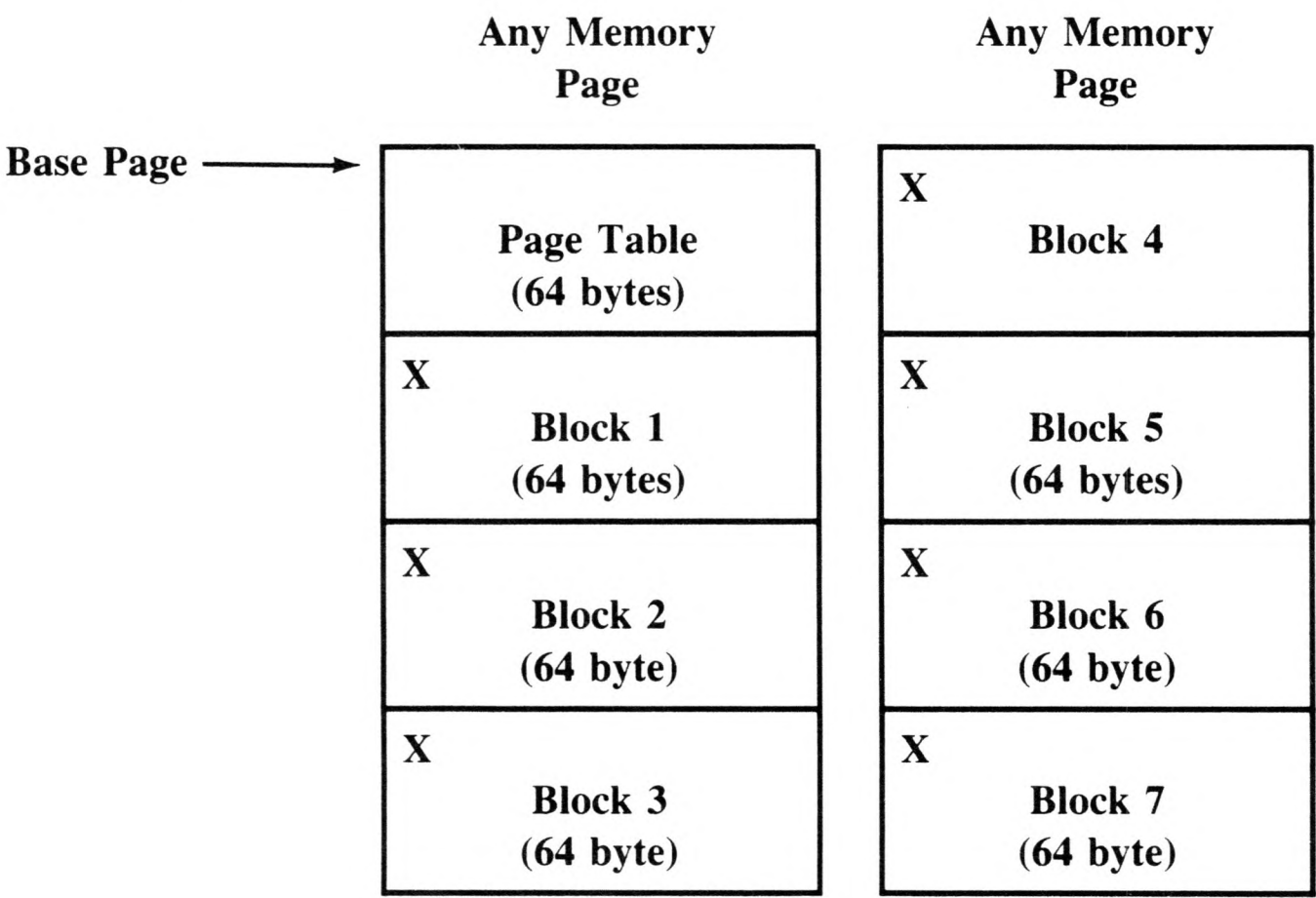
Function:

Dynamically allocates 64-byte blocks of memory by splitting pages (256 bytes) into four sections.

OS-9 uses the first 64 bytes of the base page as a “page table.” This table contains the MSB of all pages in the memory structure. If Register X passes a value of zero, the call allocates a new base page and the first 64-byte memory block.

Whenever a new page is needed, a Request Sysmem system call (F\$srqmem) executes automatically. The first byte of each block contains the block number. Routines that use Allocate 64 call should not alter this byte.

The diagram below shows how seven blocks might be allocated:



Allocate 64 is a privileged system mode call.

Entry Conditions:

X = *base address of the page table*; 0 = the page
 table has not been allocated

Exit Conditions:

A = *block number*
X = *base address of the page table*
Y = *address of the block*

If error:

CC = *C bit set*
B = *error code*

Find 64

OS9 F\$find64

103F 2F

Function:

Returns the address of a 64-byte memory block.

OS-9 used Find 64 to find process descriptors and path descriptors when given their block number. The block number may be any positive integer.

Find 64 is a privileged system mode call.

Entry Conditions:

X = *address of the block*
A = *block number*

Exit Conditions:

Y = *address of the block*

If error:

CC = C bit set
B = *error code*

Function:

Deletes the specified I/O module from the system, if the module is not in use.

This system call is used mainly by IOMAN and may be of limited or no use for other applications.

I/O Delete is a privileged system mode call.

Entry Conditions:

X = *address of an I/O module*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Technical Function:

1. Register X passes the address of a device descriptor module, device driver module, or file manager module.
2. OS-9 searches the device table for the address.
3. If OS-9 finds the address, it checks the module's use count. If the count is zero, the module is not being used; OS-9 deletes it. If the count is not zero, the module is being used; OS-9 returns an error.

I/O Queue

OS9 F\$ioqu

103F 2B

Function:

Inserts the calling process into the I/O queue of the specified process and puts the calling process to sleep indefinitely.

It is assumed that routines associated with the specified process will send a wakeup signal to the calling process.

I/O Queue is a privileged system mode call.

Entry Conditions:

A = *process number*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Insert Process

OS9 F\$aproc

103F 2C

Function:

Inserts a process into the active process queue so that the process may be scheduled for execution.

All processes already in the queue are sorted by process “age.” Age is a count of how many process switches have occurred since the process’s last time slice. When a process is moved to the active process queue, its age is set according to its priority — the higher the priority, the higher the age.

An exception is a newly active process that was deactivated while in the system state. Such a process is given higher priority because it usually is executing critical routines that affect shared system resources.

Insert Process is a privileged system mode call.

Entry Conditions:

X = *address of the process descriptor*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

Next Process

OS9 F\$npoc

103F 2D

Function:

Takes the next process out of the active process queue and executes it.

If there is no process in the queue, OS-9 waits for an interrupt, and then checks the queue again.

Next Process is a privileged system mode call.

Entry Conditions:

None

Exit Conditions:

Control does not return to caller.

Request Sysmem

OS9 F\$srqmem

103F 28

Function:

Allocates a block of memory of the specified size from the top of available RAM. The size request is rounded to the next page boundary.

Request Sysmem is a privileged system mode call.

Entry Conditions:

D = *byte count*

Exit Conditions:

U = *starting address of the memory area*

If error:

CC = C bit set

B = *error code*

Return 64

OS9 F\$ret64

103F 31

Function:

Deallocates a 64-byte block of memory. (See the Allocate 64 system call for more information.)

Return 64 is a privileged system mode call.

Entry Conditions:

X = *address of the base page*
A = *block number*

Exit Conditions:

None

If error:

CC = C bit set
B = *error code*

Return Sysmem

OS9 F\$srtmem

103F 29

Function:

Deallocates a block of contiguous pages.

Return Sysmem is a privileged system mode call.

Entry Conditions:

U = *starting address of memory to return; must point to an even page boundary*

D = *number of bytes to return*

Exit Conditions:

None

If error:

CC = C bit set

B = *error code*

SET IRQ

OS9 F\$irq

103F 2A

Function:

Adds a device to or removes it from the IRQ polling table.

This system call is used mainly by device driver routines. (See “Interrupt Processing” in Chapter 2 for a complete discussion of the interrupt polling system.)

Packet Definitions:

Flip Byte	Selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).
Mask Byte	Selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.
Priority	The device priority number: 0 = lowest, 255 = highest

Set IRQ is a privileged system mode call.

Entry Conditions:

X =	0 to remove a device, or
X =	<i>address of a packet</i> to add a device
[x]	= flip byte
[x + 1]	= mask byte
[X + 2]	= priority
U	= <i>address of the service routine's memory area</i>
Y	= <i>address of the device IRQ service routine</i>
D	= <i>address of the device status register</i>

Exit Conditions:

None

If error:

CC	= C bit set
B	= error code

Verify Module

OS9 F\$vm modul

103F 2E

Function:

Checks the module header parity and CRC bytes of a module.

If these values are valid, OS-9 searches the module directory for a module that has the same name. If one exists, OS-9 keeps the module that has the higher revision level.

Verify Module is a privileged system mode call.

Entry Conditions:

X = *address of the module to verify*

Exit Conditions:

U = *address of the module directory entry*

If error:

CC = *C bit set*

B = *error code*

Appendix A / Alphabetical System Call Lists

User and I/O System Calls

Name	Call	Purpose	Code
Allocate Bits	F\$allbit	Allocate in a bit map	13
Attach	I\$attach	Attach a new device to the system	80
Chain	F\$chain	Load and execute a new primary module	05
Change Directory	I\$chgdir	Change the working directory	86
Close Path	I\$close	Close a path to a file/device	8F
Compare Names	F\$cmpnam	Compare two names	11
CRC	F\$crc	Compute the CRC	17
Create File	I\$create	Create a path to a new file	83
Deallocate Bits	F\$delbit	Deallocate in a bit map	14
Delete File	I\$delete	Delete a file	87
Detach Device	I\$detach	Remove a device from the system	81
Duplicate Path	I\$dup	Duplicate a path	82
Exit	F\$exit	Terminate the calling process	06
Fork	F\$fork	Create a new process	03
Get ID	F\$id	Get a process ID / user ID	0C
Get Status	I\$getstt	Get file/device status	8D
Intercept	F\$icpt	Set up a signal intercept trap	09
Link	F\$link	Link to a memory module	00
Load	F\$load	Load module(s) from a file	01
Make Directory	I\$mkdir	Make a new directory	85
Memory	F\$mem	Set the memory size	07
Open Path	I\$open	Open a path to a file/device	84
Parse Name	F\$prsnam	Parse a path name	10
Print Error	F\$perr	Print error message	0F
Read	I\$read	Read data from a file/device	89
Read Line	I\$readln	Read a text line with editing	8B
Search Bits	F\$schbit	Search the bit map for a free area	12

User and I/O System Calls

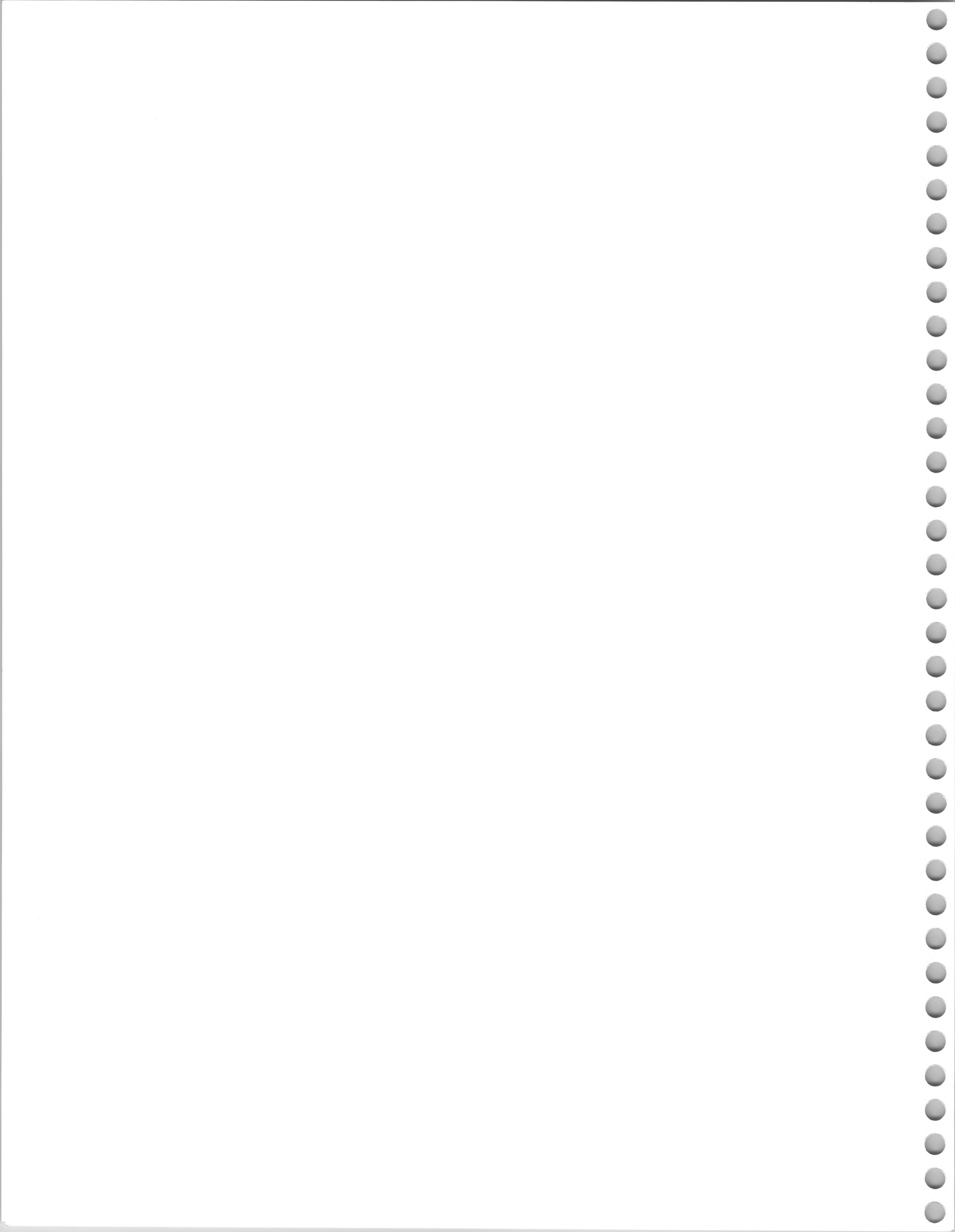
Name	Call	Purpose	Code
Seek	I\$seek	Reposition the logical file pointer	88
Send	F\$send	Send a signal to another process	08
Set Priority	F\$sprior	Set the process priority	0D
Set Status	I\$setstt	Set the file/device status	8E
Set SVC	F\$ssvc	Install a function request	32
Set SWI	F\$sswi	Set an SWI vector	0E
Set Time	F\$time	Set the system date and time	16
Sleep	F\$sleep	Put the calling process to sleep	0A
Time	F\$time	Get the system date and time	15
Unlink	F\$unlink	Unlink a module	02
Wait	F\$wait	Wait for a child process to die	04
Write	I\$write	Write to a file/device	8A
Write Line	I\$writln	Write a line of text with editing	8C

System Mode System Calls

Name	Call	Purpose	Code
Allocate 64	F\$all64	Allocate a 64-byte memory	30
Find 64	F\$find64	Find a 64-byte memory block	2F
I/O Delete	F\$iodel	Delete an I/O device from the system	33
I/O Queue	F\$ioqu	Insert a process into the I/O queue	2B
Insert Process	F\$aproc	Insert a process into the active process queue	2C
Next Process	F\$nproc	Start the next process	2D
Request Systemem	F\$srqmem	Request system memory	28

System Mode System Calls

Code	Name	Call	Purpose	
Return 64		F\$ret64	Deallocate (return) a 64-byte memory block	31
Return Sysmem		F\$srtmem	Return system memory	29
Set IRQ		F\$irq	Add or remove a device from the IRQ tables	2A
Verify Module		F\$vmodul	Verify a module	2E



Appendix B / Numerical System Call Lists

User and I/O System Calls

Code	Name	Call	Purpose
00	Link	F\$link	Link to a memory module
01	Load	F\$load	Load module(s) from a file
02	Unlink	F\$unlink	Unlink a module
03	Fork	F\$fork	Create a new process
04	Wait	F\$wait	Wait for a child process to die
05	Chain	F\$chain	Load and execute a new primary module
06	Exit	F\$exit	Terminate the calling process
07	Memory	F\$mem	Set the memory size
08	Send	F\$send	Send a signal to another process
09	Intercept	F\$icpt	Set up a signal intercept trap
0A	Sleep	F\$sleep	Put the calling process to sleep
0C	Get ID	F\$id	Get a process ID / user ID
0D	Set Priority	F\$sprior	Set the process priority
0E	Set SWI	F\$sswi	Set an SWI vector
0F	Print Error	F\$perr	Print error message
10	Parse Name	F\$prsnam	Parse a path name
11	Compare Names	F\$cmpnam	Compare two names
12	Search Bits	F\$schbit	Search the bit map for a free area
13	Allocate Bits	F\$allbit	Allocate in a bit map
14	Deallocate Bits	F\$delbit	Deallocate in a bit map
15	Time	F\$time	Get the system date and time
16	Set Time	F\$stime	Set the system date and time
17	Crc	F\$crc	Compute the CRC
32	Set SVC	F\$ssvc	Install a function request
80	Attach Device	I\$attach	Attach a new device to the system
81	Detach Device	I\$detach	Remove a device from the system

User and I/O System Calls

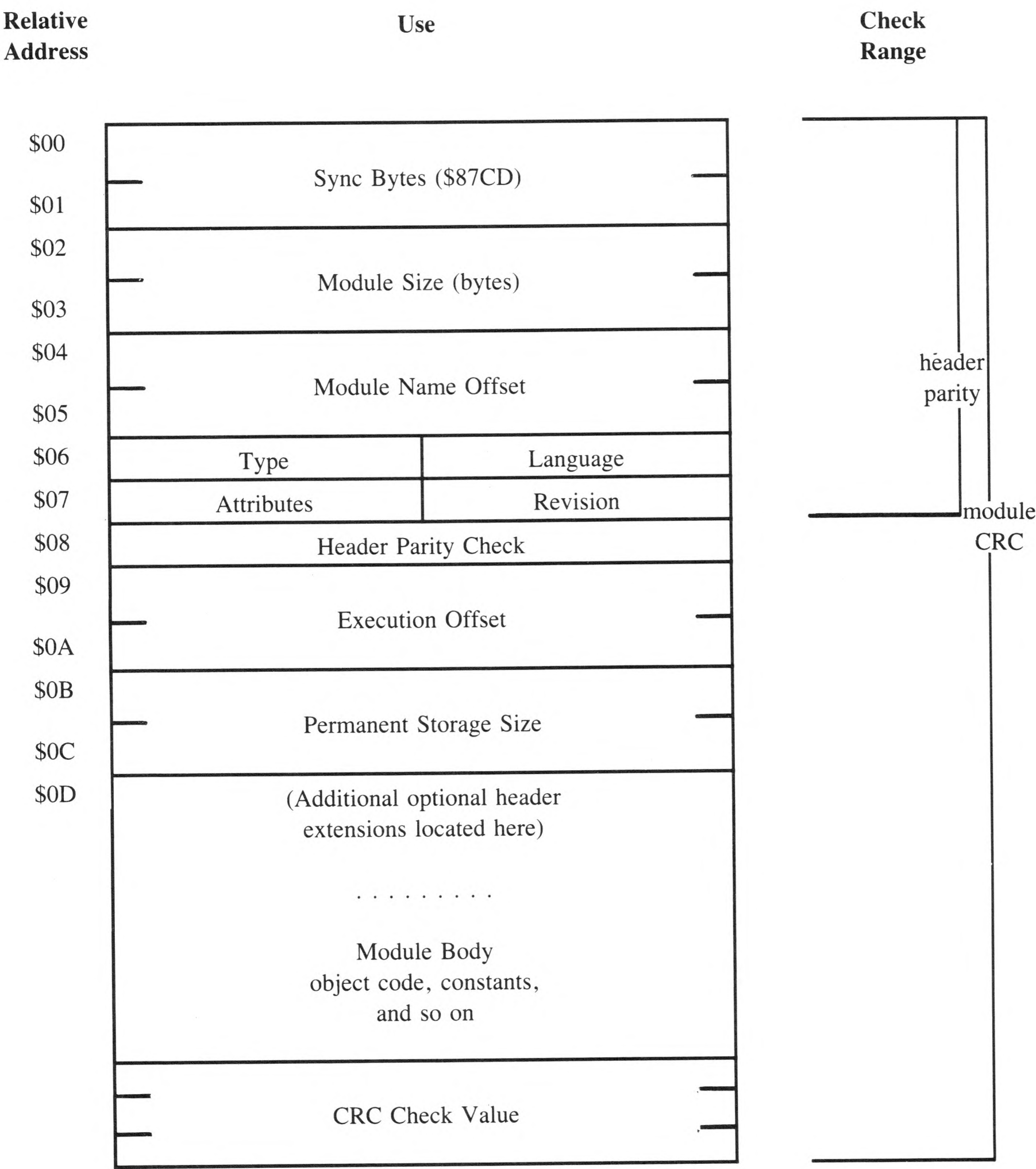
Code	Name	Call	Purpose
82	Duplicate Path	I\$dup	Duplicate a path
83	Create File	I\$create	Create a path to a new file
84	Open Path	I\$open	Open a path to a file/device
85	Make Directory	I\$mkdir	Make a new directory
86	Change Directory	I\$chgdir	Change the working directory
87	Delete File	I\$delete	Delete a file
88	Seek	I\$seek	Reposition the logical file pointer
89	Read	I\$read	Read from a file/device
8A	Write	I\$write	Write to a file/device
8B	Read Line	I\$readln	Read a text file with editing
8C	Write Line	I\$writln	Write a line of text with editing
8D	Get Status	I\$getstt	Get the file/device status
8E	Set Status	I\$setstt	Set the file/device status
8F	Close Path	I\$close	Close a path to a file/device

System Mode System Calls

Code	Name	Call	Purpose
2A	Set IRQ	F\$irq	Add or remove a device from the IRQ tables
2B	I/O Queue	F\$ioqu	Insert a process into the I/O queue
2C	Insert Process	F\$aproc	Insert a process into the active process queue
2D	Next Process	F\$nproc	Start the next process
2E	Verify Module	F\$vmodul	Verify a module
2F	Find 64	F\$find64	Find a 64-byte memory block
28	Request Sysmem	F\$srqmem	Request system memory
29	Return Sysmem	F\$srtmem	Return system memory
30	Allocate 64	F\$all64	Allocate a 64-byte memory block
31	Return 64	F\$ret64	Deallocate (return) a 64-byte memory block
33	I/O Delete	F\$iodel	Delete an I/O device from the system

Appendix C / Memory Module Diagrams

Executable Memory Module Format



Device Descriptor Format

Relative Address	Use	Check Range
\$00	Sync Bytes (\$87CD)	header parity
\$01		
\$02	Module Size	
\$03		
\$04	Offset to Module Name	
\$05		
\$06	\$F (Type)	
\$07	Attributes	
\$08	Header Parity Check	
\$09	Offset to File Manager Name String	module CRC
\$0A		
\$0B	Offset to Device Driver Name String	
\$0D	Mode Byte	
\$0E	Device Controller	
\$0F	Absolute Physical Addr. (24 bit)	
\$10		
\$11	Initialization Table Size	
\$12,\$12 + n	(Initialization Table)	
	(Name Strings, and so on)	
	CRC Check Value	

INIT Module Format

Relative Address	Use	Check Range
\$00	Sync Bytes (\$87CD)	
\$01		
\$02	Module Size (bytes)	
\$03		
\$04	Module Name Offset	header parity
\$05		
\$06	\$F (Type)	\$1 (Lang)
\$07	Attributes	Revision
\$08	Header Parity Check	
\$09	Forced Limit of Top of Free RAM	module CRC
\$0A		
\$0B		
\$0C	#IRQ Polling Table Entries	
\$0D	#Device Table Entries	
\$E0	Offset to Startup Module Name String	
\$0F		
\$10	Offset to Default Mass Storage Device Name String	
\$11		
\$12	Offset to Bootstrap Module Name String	
\$13		
\$14-n	Name Strings	
	CRC Check Value	

Appendix D / Standard Floppy Disk Format

TRS-80 Color Computer

Physical Track Format Pattern

Format	Bytes (Dec)	Value (Hex)
Header pattern (once per track)	32	4E
	12	00
	3	F5
	1	FC
	32	4E
Sector pattern (repeated 18 times)	8	00
	3	F5
	1	tracknumber (0-34)
	1	side number (0)
	1	sector number (1-18)
	1	sector length code (1)
	2	CRC
	22	4E
	12	00
	3	F5
	1	FB
	256	data area
	2	CRC
	24	4E
Trailer pattern (once per track)	N	4E (fill to index mark)

Appendix E / System Call Error Codes

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

Dec	Hex	Message	Meaning
200	\$C8	PATH TABLE FULL	The file cannot be opened because the system path table is full.
201	\$C9	ILLEGAL PATH NUMBER	The number is too large or is for a nonexistent path.
202	\$CA	INTERRUPT POLLING TABLE FULL	You can't add an interrupt.
203	\$CB	ILLEGAL MODE	You tried to perform an I/O function of which the device or file is incapable.
204	\$CC	DEVICE TABLE FULL	You can't add a device.
205	\$CD	ILLEGAL MODULE HEADER	The module is not loaded; its sync code, header parity, or CRC is incorrect.
206	\$CE	MODULE DIRECTORY FULL	You can't add a module.
207	\$CF	MEMORY FULL	There is not enough contiguous RAM free.
208	\$D0	ILLEGAL SERVICE REQUEST	The system call contains an illegal code number.
209	\$D1	MODULE BUSY	The module is non-sharable and is being used by another process.
210	\$D2	BOUNDARY ERROR	The memory allocation or deallocation request is not on a page boundary.
211	\$D3	END OF FILE	The end of the file was encountered on a read.
212	\$D4	NOT YOUR MEMORY	You tried to deallocate memory that was not assigned.
213	\$D5	NON-EXISTING SEGMENT	The device's file structure is damaged.
214	\$D6	FILE NOT ACCESSIBLE	The file attributes do not permit the access requested.
215	\$D7	BAD PATH NAME	The pathlist contained a syntax error — possibly an illegal character.

Dec	Hex	Message	Meaning
216	\$D8	FILE NOT FOUND	The specified pathlist does not exist.
217	\$D9	SEGMENT LIST FULL	The file is too fragmented to be expanded.
218	\$DA	FILE ALREADY EXISTS	The specified filename already exists in the current directory.
219	\$DB	ILLEGAL BLOCK ADDRESS	The device's file structure is damaged.
220	\$DC	ILLEGAL BLOCK SIZE	The device's file structure is damaged.
221	\$DD	MODULE NOT FOUND	You tried to link to a module that is not in the directory.
222	\$DE	SECTOR OUT OF RANGE	The device's file structure is damaged or incorrectly formatted.
223	\$DF	SUICIDE ATTEMPT	You tried to return the memory that contains your stack.
224	\$E0	ILLEGAL PROCESS ID NUMBER	The specified process does not exist.
226	\$E2	NO CHILDREN	The process can't do a Wait because it has no children.
227	\$E3	ILLEGAL SWI CODE	The code must be from 1 to 3.
228	\$E4	KEYBOARD ABORT	The process was aborted by Signal Code 2.
229	\$E5	PROCESS TABLE FULL	The process can't do a fork now.
230	\$E6	ILLEGAL PARAMETER AREA	The high and low bounds passed in Fork call are incorrect.
231	\$E7	KNOWN MODULE	This module is for internal use only.
232	\$E8	INCORRECT CRC	The module has a bad CRC value.
233	\$E9	SIGNAL ERROR	The receiving process has a previous, unprocessed signal pending.
234	\$EA	NONEXISTENT MODULE	The specified module does not exist.

Dec	Hex	Message	Meaning
235	\$EB	BAD NAME	The name uses an illegal syntax.
236	\$EC	BAD HEADER	The module header parity is incorrect.
237	\$ED	RAM FULL	There is no free system RAM at this time.
238	\$EE	BAD PROCESS ID	The process ID is incorrect.
239	\$EF	NO TASK NUMBER AVAILABLE	All task numbers are in use.
240	\$F0	UNIT ERROR	The device unit does not exist.
241	\$F1	SECTOR ERROR	The sector number is out of range.
242	\$F2	WRITE PROTECT	The device is write protected.
243	\$F3	CRC ERROR	A CRC error occurred on a read or write verify.
244	\$F4	READ ERROR	A data transfer error occurred during a disk read or an SCF (terminal) input buffer overrun.
245	\$F5	WRITE ERROR	A hardware error occurred during a disk write.
246	\$F6	NOT READY	The device has the “not ready” status.
247	\$F7	SEEK ERROR	You attempted a physical seek to a nonexistent sector.
248	\$F8	MEDIA FULL	The media does not contain enough free space.
249	\$F9	WRONG TYPE	You tried to read an incompatible media, such as a double-sided disk on a single-sided drive.
250	\$FA	DEVICE BUSY	A non-sharable device is in use.



Appendix F / Module and I/O Attributes

Standard I/O Paths

\$00 = Standard Input
\$01 = Standard Output
\$02 = Standard Error Output

Module Languages

\$00 = Data
\$01 = 6809 Object Code
\$02 = BASIC 09 I-Code
\$03 = Pascal P-Code
\$04 = Cobol I-Code

File Access Codes

\$01 = Read
\$02 = Write
Read + Write = Update
\$04 = Exec
\$08 = PRead
\$10 = PWrite
\$20 = PExec
\$40 = Share
\$80 = Dir

Module Types

\$01 = Program Module
\$02 = Subroutine Module
\$03 = Multi Module
\$04 = Data Module
\$0C = System Module
\$0D = File Manager
\$0E = Device Driver
\$0F = Device Descriptor

Module Attributes

\$8 = Reentrant

INDEX

A

- Access codes 111, 167
- Active process 11,12
- Active state 11
- Age 12
- Allocate 64 (F\$all64) 136-137
- Allocation bit map 8
 - see Bit map
- Assembly-language programming 67-73
 - Addressing variables and data structures 68
 - Extended addressing 68
 - Interrupt-driven device drivers, parts 70
 - Interrupt-driven device drivers, writing 70-71
 - Interrupt masks 69
 - Position-independent code, how to write 67
 - Rules for writing 67
 - Sample program 72-73
 - Stack requirements 69
 - Standard I/O paths, using 69
- Attach (I\$attach) 80

B

- Bit map 8
 - Clearing bits in (F\$delbit) 90
 - Searching for a free block (F\$schbit) 118
- Boot 3
- Bootstrap module 3
 - see Boot

C

- Chain (F\$chain) 82-83
- Change Directory (I\$chgdir) 84
- Character-by-character I/O (SCFMAN) 2, 27, 53-65
- Child process 10
 - Creating (F\$fork) 95
- Clearing bits in allocation bit map (F\$delbit) 90
- Clock module 2
- Close Path (I\$close) 85
- Commands, interpreting (Shell) 3
- Compare Names (F\$cmpnam) 86

INDEX

- CRC (F\$CRC) 87
- CRC byte, checking (F\$vm modul) 147
- CRC value 87
- Create File (I\$create) 88
- Cyclic redundancy count, calculating (F\$crc) 87
- CWAI instruction 12

D

- Date 129, 131
 - Returning (F\$time) 131
 - Setting (F\$time) 129
- Deallocate bits (F\$delbit) 90
- Delete File (I\$delete) 91
- Detach Device (I\$detach) 92
- Device 80, 92, 99-102, 111-112, 122-125
 - Adding to or removing from IRQ polling table (F\$irq) 146
 - Attaching a new (I\$attach) 80
 - Detaching (I\$detach) 92
 - Opening a path to (I\$open) 111-112
 - Parameters, handling (I\$getstt and I\$setstt) 99-102, 122-125
 - Status, returning (I\$getstt) 99-102
 - Status, setting (I\$setstt) 122-125
 - Table 26
 - Verifying (I\$attach) 80
 - Writing to (I\$write) 134
 - Writing to with editing (I\$writeln) 135
- Device descriptor module 3, 28-30, 39-40, 57-58
 - Format of 30, 158
- Device driver module 3, 27-28, 40-41, 58
 - Branch table 28
- Directory 37
 - Changing the working directory (I\$chkdir) 84
 - Creating and initializing (I\$mkdir) 108
- Disk allocation map sector (LSN 1) 34
- Disk file
 - see File
- Disk I/O (RBFMAN) 2, 27, 33-52

INDEX

E

Editing

 see Line Editing Functions 53-54

Error message 115, 163-165

 Writing (F\$perr) 115

Exit (F\$exit) 94

F

File 2, 8, 14, 26-27, 35-35, 88, 91, 99-102, 111-112, 122-125, 135

 Creating and opening (I\$create) 88

 Deleting (I\$delete) 91

 Opening a path to (I\$open) 11-112

 Pointer, repositioning (I\$seek) 119

 Status, returning (I\$gttstt) 99-102

 Status, setting (I\$setstt) 122-125

 Terminating path to (I\$close) 85

 Writing to (I\$write) 134

 Writing to with editing (I\$writln) 135

File access codes 111

File descriptor sector 35-36

File manager module 2, 26-27

 see also RBFMAN, SCFMAN, and PIPEMAN

Find 64 (F\$find64) 138

Firq interrupt 14

Fork (F\$fork) 10, 95

Free block, searching bit map for (F\$schbit) 118

Free memory 8

Function calls 7, 75

G

Get ID (F\$id) 98

Get Status (I\$gttstt) 99-102

INDEX

H

Header 19-21

- Sync Bytes 19
- Module size 19
- Offset to module name 20
- Type/language byte 20
- Attributes/revision byte 21
- Header check 21
- Execution offset 23
- Permanent storage size 23

Hexadecimal number iv

I

ID, returning (F\$id) 98

Identification sector (LSN 0) 34

INIT 2, 5, 159

Input/output manager 2, 26

see IOMAN

Insert Process (F\$aproc) 141

Intercept (F\$icpt) 103

Intercept trap, setting (F\$icpt) 103

Interrupt processing 14-16

Interrupt vectors, setting for SWI2 and SWI3

instructions

(F\$sswi) 128

IOMAN 2, 26

I/O 2-3, 25, 27-28, 40-41, 53-65, 69

Character-by-character (SCFMAN) 2, 27, 53-65

Modules 25

Non-disk (SCFMAN) 2, 27, 53-65

Paths, standard 69, 167

Physical functions for specific I/O controller

hardware (device driver module) 3, 27-28,
40-41, 58

System calls 6, 75, 149-150, 153-154

I/O Delete (F\$iodel) 139

I/O port, associating with its logical name, device driver
and file manager (device descriptor module) 3,
28-30, 39-40, 57-58

I/O Queue (F\$ioqu) 140

IRQ interrupt 15

INDEX

IRQ polling table 15-16

Adding to or removing from (F\$irq) 16, 146

Flip byte 15

Mask byte 15

Polling address 15 Priority 16

Service routine address 16

Static storage address 16

K

Kernel 2, 5-16

Functions of 5

L

Line editing functions 53-54

Link (F\$link) 104-105

Load (F\$load) 106-107

Logical interrupt polling system 15-16

Logical sector number 33

LSN 0 34

LSN 1 35

M

Make Directory (I\$mkdir) 108

Memory 110

Allocating automatically 8

Allocating block from top of available RAM
(F\$srqmem) 143

Deallocating block of contiguous pages
(F\$srtmem) 145

Deallocating 64-byte block (F\$ret64) 144

Dynamically allocating 64-byte blocks (F\$all64)
136-137

Size, setting (F\$mem) 110

Memory (F\$mem) 110

Memory management, advantages of 7

Memory map 9

Memory module iii, 1

see Module

INDEX

Module iii, 1

- Attributes 17, 167
 - Deleting an I/O module (F\$iodel) 139
 - Executable memory module format 22, 157
 - Format (parts) 18-19
 - Languages 20, 167
 - Linking to (F\$link) 104-105
 - Loading and executing (F\$chain) 82-83
 - Loading from a file (F\$load) 106-107
 - Loading from the working executing directory (F\$load)
106-107
 - ROM 23-24
 - Types 17, 167
 - Unlinking (F\$unlink) 132
- Module body 19
- Module CRC value 19
- Module header 18
- See Header
- Multiprogramming 9-13

N

- Name 86, 113-114
- Analyzing a text string for (F\$prsnam) 113-114
 - Comparing names (F\$cmpnam) 86
- Next Process (F\$nproc) 142
- NMI interrupt 15
- Non-disk I/O (SCFMAN) 2, 27, 53-65

O

- Open Path (I\$open) 111-112
- OS9Defs 75

P

- Page 8
- Parity byte, checking (F\$vmodul) 147
- Parse Name (F\$prsnam) 113-114

INDEX

Path

- Duplicating (I\$dup) 93
- Opening (I\$open) 111-112
- Reading from (I\$read and I\$readln) 116,117
- Repositioning its logical file pointer (I\$seek) 119
- Terminating (I\$close) 85

- Path descriptor (PD) 31-32, 37-38, 54-56
 - Standard information 31-32

Path table 26

Pipe file manager 2

PIPEMAN 2

Pipes 2-3

Pointer, repositioning (I\$seek) 119

Primary module 10

Print Error (F\$perr) 115

Priority 16

- Setting (F\$sprior) 121

Process 9

- Creating a new (F\$fork) 10-11
- Inserting into I/O queue and putting to sleep (F\$ioqu) 140
- Inserting into active queue for execution scheduling (F\$aproc) 141
- Removing from active process queue and executing (F\$nproc) 142
- Terminating the calling process (F\$exit) 11, 94
- Turning off the calling process until child process dies (F\$wait) 133
- Turning off the calling process temporarily (F\$sleep) 130

Process age 12

Process descriptor 10

Process ID 11

- Returning (F\$id) 98

Process priority, setting (F\$sprior) 121

Process states 11-12

Q

Queue 2

INDEX

R

- Random Block File Manager 2, 27, 33-52
 - see RBFMAN
- RBFDefs 32
- RBFMAN 2, 27, 3-52
- Read (I\$read) 116
- Read Line (I\$readln) 117
- Real-time clock 10
 - Starting (F\$time) 129
- Request System (F\$srqmem) 143
- Return System (F\$srtmem) 145
- ROM modules 23-24
- Root directory 37
- RTI instruction 13, 16
- RTS instruction 13, 16

S

- SCFDefs 32
- SCFMAN 2, 27, 53-65
 - Line editing functions 53-54
- Search Bits (F\$schbit) 118
- Seek (I\$seek) 119
- Send (F\$send) 13, 120
- Sequential Character File Manager 2, 27, 53-65
 - see SCFMAN
- Set IRQ (F\$irq) 146
- Set Priority (F\$sprior) 121
- Set SVC (F\$ssvc) 126-127
- Set Status (I\$setstt) 122-125
- Set SWI (F\$sswi) 128
- Set Time (F\$time) 129
- Shell 3
- Sleep (F\$sleep) 12, 130
- Sleeping state 12
- Signal 12-13
 - Codes 13
 - Intercept trap, setting (F\$icpt) 13
 - Sending (F\$send) 120
- String
 - Analyzing for legal OS-9 name (F\$prsnam) 113-114
 - Comparing strings (F\$cmpnam) 86

INDEX

- SWI interrupt 14
- SWI2 interrupt 14
 - Setting interrupt vectors for (F\$sswi) 128
- SWI3 interrupt 14
 - Setting interrupt vectors for (F\$sswi) 128
- SYSGO 5
- System call 6, 75
 - Adding or replacing (F\$ssvc) 126-127
 - Lists 75, 149-155
 - Processing 5, 6-7, 76-77
 - Types 6-7
- System initialization 5
- System mode calls 76
- System startup task (SYSGO) 5

T

- Tick 10
- Time 10, 19, 131
 - Returning (F\$time) 131
 - Setting (F\$time) 129
 - Slice 10

U

- Unix operating system iii
- Unlink (F\$unlink) 132
- User calls 149-150, 153-154
- User ID 11
 - Returning (F\$id) 98

V

- Vectors 14
 - Hardware 14
 - Setting for SWI2 and SWI3 instructions (F\$sswi) 14, 128
- Verify Module (F\$vm modul) 147

INDEX

W

Wait (F\$wait) 133
Waiting state 5, 11
Western Digital Floppy Disk Controller IC 39
Wildcard calls 99-102, 122-125
Working directory 84
 Changing (I\$chkdir) 84
Write (I\$write) 134
Write Line (I\$writeln) 135

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA

**91 KURRAJONG ROAD
MOUNT DRUITT, N.S.W. 2770**

BELGIUM

**PARC INDUSTRIEL DE NANINNE
5140 NANINNE**

U. K.

**BILSTON ROAD WEDNESBURY
WEST MIDLANDS WS10 7JN**